UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Max Ruben de Oliveira Schultz

GERAÇÃO AUTOMÁTICA DE FERRAMENTAS DE INSPEÇÃO DE CÓDIGO PARA PROCESSADORES ESPECIFICADOS EM ADL

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Olinto José Varela Furtado, Dr.
Orientador

Prof. Luiz Cláudio Villar dos Santos, Dr.
Co-Orientador

Florianópolis, março de 2007

GERAÇÃO AUTOMÁTICA DE FERRAMENTAS DE INSPEÇÃO DE CÓDIGO PARA PROCESSADORES ESPECIFICADOS EM ADL

Max Ruben de Oliveira Schultz

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação, e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Rogério Cid Bastos, Dr.
(coordenador)

Banca Examinadora

Prof. Olinto José Varela Furtado, Dr.
Orientador

Prof. Luiz Cláudio Villar dos Santos, Dr.
Co-Orientador

Prof. Luís Fernando Friedrich, Dr.

Prof. José Luís Almada Güntzel, Dr.

"A tecnologia ensinou uma coisa à humanidade: nada é impossível."
(Lewis Mumford)

Às pessoas mais importantes da minha vida, Michele, minha noiva, esposa, mulher, amiga e companheira, meus pais Rubin e Zélia e minha irmã Rúbia, que sempre estiveram me apoiando ao longo desta caminhada.

Agradecimentos

Minha vida é feita de metas e objetivos a serem alcançados e neste momento desejo partilhar minha felicidade e agradecimentos:

Ao professor Omero Francisco Bertol da UTFPR - Campus Pato Branco/PR pelo incentivo e apoio para o meu ingresso na área de docência.

Aos professores Olinto J. V. Furtado e Luiz Cláudio Villar dos Santos pela oportunidade de cursar o mestrado, um sonho que buscava há três anos, e também por toda orientação e direcionamento prestados, para mim, profissionais exemplares.

Ao Alexandro Baldassin do Laboratório de Sistemas de Computação da UNICAMP por todo o suporte técnico e atenção dispensada.

Aos colegas e amigos do laboratório que contribuiram de alguma forma na implementação para conclusão deste trabalho, Paulo F. Kuss, Alexandre K. I. Mendonça, Felipe G. Carvalho, Daniel C. Casarotto e José O. C. Filho.

Agradeço também as contribuições de todos os membros da banca.

Max Ruben de Oliveira Schultz

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Acrônimos

ADL *Architecture Description Language*

ASIP *Application-Specific Instruction-Set Processor*

BFD *Binary File Descriptor Library*

BNF *Backus-Naur Form*

CISC *Complex Instruction Set Computer*

COFF *Common Object File Format*

CPU *Central Processing Unit*

DSP *Digital Signal Processor*

ELF *Executable and Linkable Format*

GLC Gramática Livre de Contexto

GNU *Gnu is not Unix*

GPP *General Purpose Processor*

IPs *Intelectual Property Blocks*

ISA *Instruction Set Architecture*

MPSoCs *Multiple-Processor SoCs*

NoC *Network-on-Chip*

RISC *Reduced Instruction Set Computer*

RT *Register-Transfer Level*

SoC *System-on-a-Chip*

TLM *Transaction-Level Modeling*

VLIW *Very Long Instruction Word*

Resumo

Um sistema embarcado pode ter todos os seus componentes eletrônicos implementados em um único circuito integrado, dando origem ao assim chamado *System-on-a-Chip* (SoC). Um SoC é composto de uma ou mais CPUs e por componentes não programáveis, tais como memória(s), barramento(s) e periférico(s). A CPU escolhida pode ser um processador dedicado, denominado *Application-Specific Instruction-Set Processor* (ASIP).

O projeto de SoCs requer ferramentas para a inspeção de código, a fim de se explorar a corretude do software embarcado a ser executado em cada CPU. Isto pode ser feito através da geração automática de ferramentas a partir de um modelo formal de CPU, cujas características podem ser descritas através do uso de Linguagens de Descrição de Arquiteturas (*Architecture Description Language* - ADLs). Como o redirecionamento manual das ferramentas para cada CPU explorada seria inviável devido à pressão do *time-to-market*, o redirecionamento automático é mandatório.

Esta dissertação contribui com a expansão do módulo de geração de ferramentas de manipulação de código binário associado à ADL ArchC, através da geração automática de desmontadores e depuradores de código.

As ferramentas de desmontagem e depuração de código foram validadas por meio de comparação com ferramentas nativas congêneres para modelos de arquiteturas RISC e CISC (i8051, MIPS, SPARC e PowerPC). Para fins de experimentação, foram usados os *benchmarks* MiBench e Dalton, evidenciando a corretude e a robustez das ferramentas.

Além disso, mostra-se a integração do gerador de desmontadores no âmbito de um tradutor binário, proposto como resultado de trabalho cooperativo (também reportado em outras duas dissertações correlatas).

Abstract

An embedded system may have all its components implemented in a single integrated circuit, giving rise to the so-called System-on-a-Chip (SoC). An SoC consists of one or more CPUs and of non-programmable components, such as memories, busses and peripherals. A chosen CPU may be an Application-Specific Instruction-Set Processor (ASIP).

The design of SoC asks for code inspection tools so as to verify the correctness of the embedded software to be run in each CPU. Since the manual retargeting of tools for each explored CPU would not be viable under the time-to-market pressure, the tools must be automatically retargeted. This can be done through automatic tool generation from a formal model of the CPU, whose properties can be described by means of an Architecture Description Language (ADL).

This dissertation contributes to the extention of the generator of binary utilities associated with the ADL ArchC by providing the automatic generation of code disassemblers and debuggers.

The generated disassembling and debugging tools were validated by comparing them to similar native tools for RISC and CISC architectures (i8051, MIPS, SPARC and PowerPC). For experimentation purposes, the benchmarks MiBench and Dalton were used to provide evidences of correctness and robustness of the generated tools.

Moreover, this dissertation shows the integration of the disassembling generator in the frame of a binary translator, which was proposed as a result of cooperative work (also reported in other two related dissertations).

Capítulo 1

Introdução

Sistemas embarcados ou *embedded systems* são sistemas de processamento de informações embutidos em um produto maior e têm como característica serem sistemas integrados de hardware e software [LEU 01]. Exemplos de aplicações de sistemas embarcados são encontrados em aeronaves, telefones celulares e centrais telefônicas, equipamentos de redes de computadores (*hubs*, *switches*, *firewalls*), automóveis, eletrodomésticos, equipamentos médicos, *videogames*, etc.

Um sistema embarcado pode ter todos os seus componentes eletrônicos implementados em um único circuito integrado, dando origem a um *System-on-a-Chip* (SoC) [MAR 03]. Um SoC é composto de uma ou mais CPUs, memórias, meios de comunicação e componentes dedicados conhecidos como blocos de propriedade intelectual (*Intelectual Property Blocks* - IPs). Os IPs executam uma tarefa específica que pode ser uma mera interface para conexão de entrada e saída ou um algoritmo implementado em hardware. CPUs, memória e IPs são conectados em um SoC através de um barramento ou de uma rede em chip (*Network-on-Chip* - NoC) [GHE 05].

A essência do projeto de sistemas embarcados consiste na implementação de um conjunto específico de funções que satisfaçam tanto os requisitos funcionais da aplicação (isto é o seu comportamento) quanto os requisitos não funcionais, tais como compatibilidade com restrições de tempo real, restrições impostas pelo tamanho de memória, consumo máximo de potência, etc. A escolha da arquitetura acaba por definir como o

projetista implementará uma determinada função, na forma de um componente dedicado, através de software ou na forma de um componente programável. A necessidade de implementações flexíveis, que possam ser alteradas de maneira mais rápida, popularizou o uso de software embarcado; além disso, projetar diretamente em hardware é mais caro e necessita de mais tempo para implementação [SV 01].

1.1 Projeto orientado a plataforma

Os requisitos cada vez mais complexos das aplicações de sistemas embarcados, a redução no ciclo de desenvolvimento dos produtos eletrônicos, a conseqüente necessidade de se aumentar o ganho de produtividade, os altos custos de engenharia não recorrente das tecnologias VLSI contemporâneas e o alto custo das máscaras para fabricação de circuitos integrados em tecnologias nanométricas são fatores que determinaram a adoção do projeto baseado em plataforma como paradigma para a concepção de SoCs. Em geral, plataformas são caracterizadas por uma família de arquiteturas, como processadores, IPs e memórias, que satisfazem um conjunto de restrições. A idéia principal do projeto orientado a plataformas é o reuso dos componentes de hardware e software a fim de reduzir tempo, esforço e custo de projeto [SV 01]. São exemplos de plataformas: PDesigner [PDE 07], SEEP [SEE 07], entre outras.

A demanda imposta por aplicações cada vez mais complexas resultou no uso de vários processadores (possivelmente de natureza distinta) em um mesmo SoC, dando origem a plataformas multiprocessadas e heterogêneas conhecidas como *multiple-processor SoCs* (MPSoCs).

Para reduzir o *time-to-market* [1] de sistemas cada vez mais complexos, tornou-se necessário recorrer a representações mais abstratas do SoC, que permitissem antecipar o desenvolvimento de software dependente de hardware. Essa necessidade deu origem a um novo estilo de projeto, denominado modelagem em nível de transações (*Transaction-Level Modeling* - TLM), para ser utilizado no desenvolvimento de plataformas.

[1] *Time-to-market* corresponde ao tempo necessário para que uma idéia seja implementada em um produto real [GHE 05].

1.2 A necessidade de técnicas redirecionáveis

Para a escolha da CPU de um SoC, são utilizados processadores de uso geral (*General Purpose Processors* - GPPs), microcontroladores, processadores digitais de sinais (*Digital Signal Processors* - DSPs) ou ainda CPUs dedicadas. Uma CPU dedicada é moldada para uma dada classe de aplicações e é por isso denominada de *Application-Specific Instruction-Set Processor* (ASIP).

Devido a esta grande variedade de CPUs, faz-se necessária a exploração de soluções alternativas. A escolha de uma CPU é parte do processo de exploração do espaço de projeto (*design space exploration*), cujo objetivo é a otimização do SoC [MAR 03].

Para se avaliar a adequação de uma dada CPU é preciso dispor de um modelo para simular seu comportamento em interação com o resto do sistema e verificar se são atendidos os requisitos da aplicação, como por exemplo a restrição ao tamanho de código (pouca memória disponível), consumo máximo de potência (demanda por produtos portáteis) e restrições de tempo real. Em suma, a exploração do espaço de projeto requer a disponibilidade de modelos de cada CPU alternativa, preferencialmente modelos executáveis. Ademais, a exploração do espaço de projeto pressupõe que se disponha de ferramentas para gerar código para cada uma das CPUs alternativas, ou seja, as ferramentas de suporte ao desenvolvimento de software precisam ser redirecionáveis para diferentes arquiteturas-alvo (*retargetable code generation*) a fim de se explorar a corretude e a eficiência do código.

Como um ASIP não é um componente padrão, não há disponibilidade imediata nem de modelos, nem de ferramentas de suporte ao desenvolvimento de seu software. Tais ferramentas e modelos devem ser geradas automaticamente, pois sua elaboração manual inviabilizaria o uso de ASIPs sob a pressão do *time-to-market*.

Por um lado, o uso de Linguagens de Descrição de Hardware (*Hardware Description Languages* - HDLs) tem se mostrado eficiente para a síntese automática do hardware de um ASIP, especialmente para descrições no nível RT (*Register-Transfer level*). Por outro lado, a eficiente geração automática de modelos e de ferramentas de suporte ao desenvolvimento de software requer o uso de Linguagens de Descrição de Arquiteturas

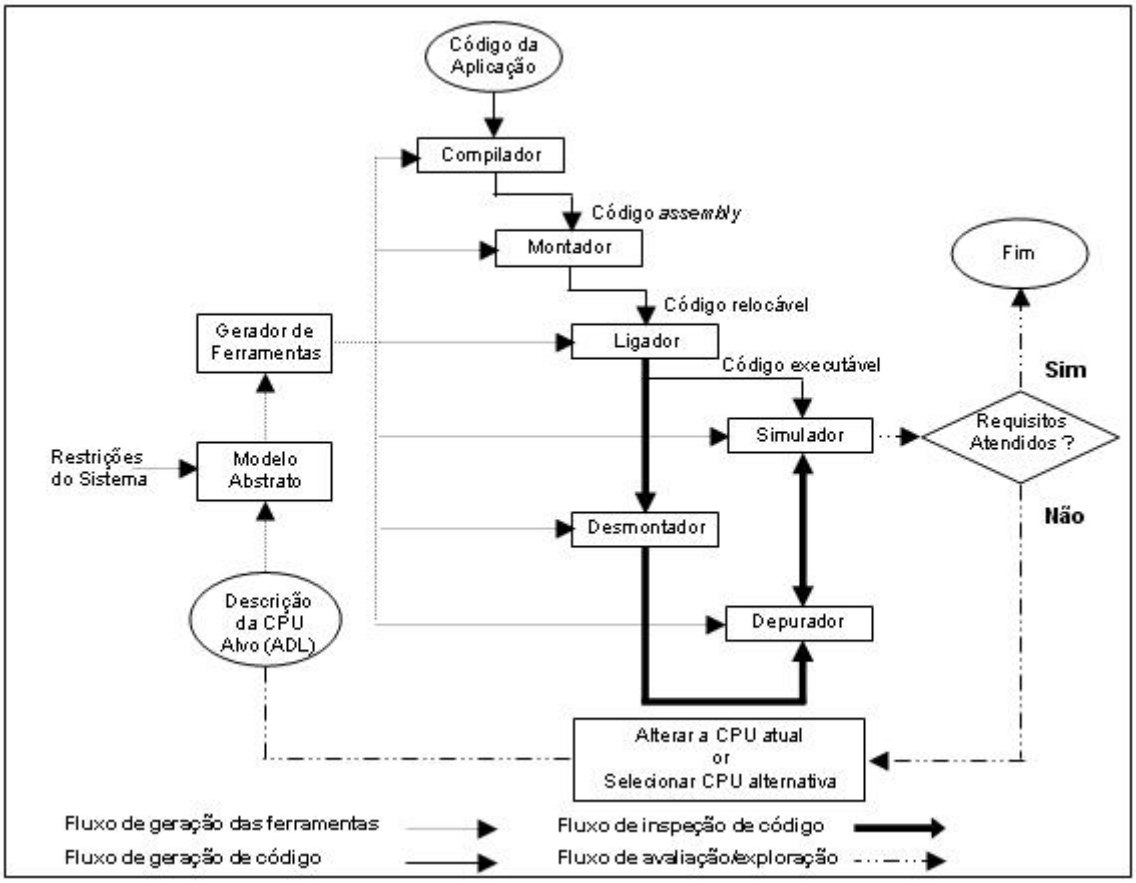

Figura 1.1: Fluxo de exploração do espaço de projeto baseado em CPUs

(*Architecture Description Language* - ADLs). Uma ADL possibilita a especificação formal de informações dependentes da arquitetura, tais como conjunto de instruções, banco de registradores, memória e tamanho da palavra de instrução [LEU 01]. Exemplos de ADLs são nML [FAU 95], Sim-nML [RAJ 98], ISDL [HAD 98], LISA [ZIV 96] e ArchC [RIG 04].

O uso de ADLs torna mais eficiente a geração automática de ferramentas de suporte ao desenvolvimento de software embarcado e facilita o projeto do hardware do ASIP [TAG 05].

Para se evitar que as ferramentas sejam dependentes de uma determinada ADL, um modelo abstrato do processador pode ser utilizado. Na prática, tal modelo deve ser sintetizável a partir de uma descrição escrita em alguma ADL. A Figura 1.1 descreve um

gerador de ferramentas baseado em um modelo abstrato, o qual ilustra diferentes classes de fluxo de informação (geração de ferramentas, geração de código, inspeção do código e avaliação do código). A utilização de um modelo abstrato torna o gerador de ferramentas independente de ADL como sugerido por Baldassin et al [BAL 07a].

A exploração consiste em quatro etapas principais conforme segue. Primeiro, dado o modelo de um processador-alvo, as ferramentas de geração de código (*backend* do compilador, montador e ligador), as ferramentas de inspeção de código (desmontador e depurador) e um simulador de instruções são automaticamente gerados. Então, o código-fonte da aplicação em linguagem C/C++ pode ser compilado, montado e ligado, resultando em um código executável. Na terceira etapa, o código é executado no simulador de instruções e sua corretude pode ser verificada com o auxílio das ferramentas de desmontagem e depuração de código. Essas ferramentas permitem que o código seja executado incrementalmente (*step-by-step*), que sejam definidos pontos de parada na execução (*breakpoints*), assim como é possível se monitorar o conteúdo de variáveis utilizadas pelo programa (*watchpoints*). Finalmente, assim que a funcionalidade apropriada for garantida com a remoção dos erros existentes, a execução contínua do código no simulador permite a avaliação da qualidade do código no que diz respeito aos requisitos do projeto. Se algum requisito não for atendido, um conjunto alternativo de instruções pode ser utilizado, a fim de se chegar a uma nova solução. Se o processador sob exploração for um ASIP, seu conjunto de instruções pode ser customizado para melhor se adequar à aplicação. Caso contrário, um outro processador pode ser selecionado como candidato a nova exploração.

1.3 A necessidade de suporte para depuração

Existe uma dificuldade em se atestar a corretude do projeto de um sistema embarcado, sendo necessária a inspeção e a avaliação do código gerado para diferentes arquiteturas-alvo. As ferramentas para criação e depuração de software para sistemas embarcados são basicamente as mesmas utilizadas para software convencional, tais como compiladores, montadores, ligadores e depuradores. A diferença está na confiabilidade e disponibilidade das mesmas [SV 01].

Além disso, uma ferramenta de depuração é também útil quando se está criando um novo modelo de um processador a partir da sua descrição em uma ADL; também é útil quando não se possui o arquivo-fonte do programa objeto e se deseja inspecionar o seu código para fins de detecção de erros, por exemplo.

O projeto de SoCs requer ferramentas para a inspeção de código em nível de sistema no âmbito de plataformas heterogêneas, com o objetivo de verificar a corretude do software embarcado a ser executado em cada CPU-alvo. Isto pode ser feito através da geração automática de ferramentas a partir de um modelo formal de CPU, cujas características podem ser descritas através do uso de *Architecture Description Language* (ADLs). Como o redirecionamento manual das ferramentas para cada CPU explorada seria inviável devido à pressão do *time-to-market*, o redirecionamento automático é mandatório [LEU 01].

1.4 A contribuição desta dissertação

Esta dissertação contribui com a expansão do pacote de ferramentas associado à linguagem ArchC [ARC 07], uma ADL contemporânea que gera modelos em SystemC [SYS 07] e é distribuída sob licença GPL [2]. Mais especificamente, a dissertação contribui para a expansão do módulo de geração de ferramentas de manipulação de código binário, denominado `acbinutils`, para permitir a geração automática, a partir dos modelos descritos, de desmontadores e depuradores de código binário, contribuindo assim, para o processo de inspeção e tradução de código.

Para garantir sua compatibilidade com outras ferramentas do módulo `acbinutils`, tais como o gerador de montadores [BAL 05] e o gerador de ligadores [CAS 06], esta dissertação seguiu as mesmas diretrizes metodológicas que nortearam aqueles trabalhos.

Essencialmente, a metodologia consiste em editar um pacote de utilitários binários de domínio público, alterando os módulos dependentes de arquitetura para fins de redirecionamento e conservando os módulos independentes de arquitetura.

[2] GPL é a Licença Pública Geral GNU (*GNU General Public License*). A formulação da GPL é tal que, ao invés de limitar a distribuição do software por ela protegido, ela apenas impede que ele seja integrado em software proprietário.

Os pacotes adotados como infraestrutura de implementação foram o GNU Binutils [PES 93] e o GNU Debugger [STA 04b]. A geração de desmontadores foi implementada através do redirecionamento da ferramenta `objdump` do primeiro pacote e a geração de depuradores foi implementada através do redirecionamento da ferramenta `gdb` do segundo pacote. Para fins de implementação do protótipo e sem perda de generalidade, adotou-se o formato *Executable and Linkable Format* (ELF) [STA 07] para codificar arquivos-objeto.

As ferramentas de desmontagem e depuração de código foram validadas por meio de comparação com as ferramentas nativas `objdump` e `gdb`, respectivamente, para diversos modelos ArchC de arquiteturas RISC e CISC disponíveis no repositório da ADL ArchC, conforme será relatado na Seção 4.4. Para fins de experimentação, foram usados os *benchmarks* MiBench [GUT 01] e Dalton [DAL 07].

As principais contribuições deste trabalho são:

- definição e formalização das características que devem ser suportadas por uma ADL para que se possa gerar automaticamente desmontadores e depuradores de código;
- proposição de uma metodologia para geração automática de desmontadores e depuradores baseados na descrição de arquiteturas em ArchC;
- integração da ferramenta de desmontagem de código no processo de tradução binária de código entre diferentes arquiteturas.

Esta dissertação está estruturada como descrito a seguir. No Capítulo 2 são abordados trabalhos correlatos com foco em geração de ferramentas de manipulação de código binário e tradução binária. O Capítulo 3 descreve a lógica de execução das ferramentas desenvolvidas através de algoritmos. A metodologia para geração automática das ferramentas redirecionáveis de inspeção de código, os resultados experimentais obtidos durante a validação das ferramentas e a formalização do conjunto de instruções de um processador são detalhados no Capítulo 4. O papel das ferramentas de inspeção de código no desenvolvimento de um tradutor de código binário é abordado no Capítulo 5. Conclusões e trabalhos futuros estão descritos no Capítulo 6. Por fim, o apêndice A traz

um manual para criação das ferramentas e o apêndice B a modelagem do modo THUMB do processador ARM.

O trabalho de pesquisa reportado nesta dissertação foi desenvolvido sob fomento parcial do Programa Nacional de Cooperação Acadêmica (PROCAD) da CAPES, no âmbito do Projeto n. 0326054, intitulado "Automação de Projeto de Sistemas Dedicados Usando uma Linguagem de Descrição de Arquiteturas", que norteia e formaliza as atividades de cooperação científica entre o Programa de Pós-Graduação em Ciência da Computação (PPGCC) da UFSC e o Instituto de Computação da UNICAMP. O projeto foi executado no Laboratório de Automação do Projeto de Sistemas (LAPS) da UFSC (http://www.inf.ufsc.br/ laps).

O desenvolvimento deste trabalho foi parcialmente amparado por bolsa de mestrado, no âmbito do Programa Nacional de Microeletrônica (PNM), Processo n. 132874 / 2005-9.

Capítulo 2

Trabalhos correlatos

Este capítulo apresenta uma visão de trabalhos e ferramentas relacionadas às técnicas utilizadas para a geração de desmontadores e depuradores; são abordadas metodologias de geração baseadas em ADL, pacotes de código redirecionável e trabalhos relacionados a tradução binária. Ao final, são destacadas as vantagens da abordagem utilizada.

2.1 Geração de ferramentas a partir de linguagens de descrição de arquiteturas (ADLs)

ADLs são concebidas especialmente para descrever CPUs. Uma ADL é uma linguagem cujas primitivas permitem a descrição do conjunto de instruções de uma CPU e, possivelmente, de algumas características e parâmetros de sua organização (comprimento da palavra de instrução, códigos operacionais, bancos de registradores, memórias, etc.), com o intuito de se gerar automaticamente ferramentas de software, como simuladores, montadores, ligadores, desmontadores, depuradores e até mesmo *backend* de compiladores [RIG 04].

Dentre as características desejáveis em uma ADL estão a possibilidade de especificação de um grande número de arquiteturas, inclusive aquelas que prevêem otimizações, e o suporte à geração de ferramentas de manipulação e geração de código [HAD 97].

A ADL nML [FAU 95] [FRE 93] [FAU 93], originalmente desenvolvida na *Techni-*

cal University of Berlin, leva em consideração informações constantes nos manuais dos processadores, tais como seu conjunto de instruções, operações de transferência entre registradores, sintaxe *assembly* e a codificação binária das instruções. A estrutura da arquitetura é descrita juntamente com o comportamento da instrução; contudo, não existem informações detalhadas sobre o bloco operativo e nem flexibilidade para essa descrição [RIG 04]. Além disso, a linguagem não fornece um mecanismo para atribuição de múltiplas sintaxes de codificações para as instruções [BAL 05].

Baseado em um modelo especificado em nML e com a inclusão de algumas informações adicionais, Hartoog et al [HAR 97] realizaram experimentos quanto a geração de ferramentas binárias, entre elas um desmontador, que é utilizado no depurador desenvolvido em linguagem C++. Também baseado em nML, a companhia Target Compiler Technologies [TEC 07] possui um simulador denominado Checkers que possui uma interface gráfica para depuração e também um gerador de código denominado Chess. Contudo, essas ferramentas não são de domínio público.

Moona [MOO 00] apresentou uma metodologia para geração de ferramentas binárias a partir da descrição do processador na ADL Sim-nML [RAJ 98], que foi desenvolvida no *Indian Institute of Technology* (IIT), a qual usa como base a nML. Nessa metodologia, destaca-se a possibilidade de geração de uma representação intermediária dos modelos descritos na ADL utilizada pelas ferramentas [RAJ 99]. Jain [JAI 99] desenvolveu um gerador de desmontadores seguindo esta metodologia e afirma que a ferramenta funciona para todos os tipos de processadores RISC e CISC. Contudo, como todo o processo de exibição da sintaxe *assembly* pelo desmontador foi implementado manualmente, sem uso de uma ferramenta redirecionável já validada, e também por terem sido apresentados resultados experimentais apenas para a CPU PowerPC603, fica a dúvida quanto à corretude e à generalidade da técnica utilizada.

Instruction Set Description Language for Retargetability (ISDL) [HAD 98], desenvolvida no *Massachussets Institute of Tecnology* (MIT), é uma ADL comportamental que se utiliza de uma gramática de atributos para descrever as instruções e foi projetada especialmente para a geração de ferramentas redirecionáveis, como desmontadores. Essa ADL permite especialmente a descrição de arquiteturas *Very Long Instruction Word* (VLIW),

e sua grande restrição é a impossibilidade da descrição de instruções multi-ciclo com tamanho variável [HAD 97].

LISA [ZIV 96] [HOF 02] é uma ADL que foi desenvolvida na *Aachen University of Technology* (RWTH) com o propósito inicial de geração automática de simuladores compilados [PEE 99]. Os elementos básicos da linguagem correspondem à declaração de recursos (constituição do hardware) e de operações (conjunto de instruções). O simulador gerado é ligado pelo usuário a uma interface gráfica de depuração para inspeção de código [HOF 01]; entretanto, não está claro qual o mecanismo de funcionamento deste processo de depuração em função da dificuldade de se obter documentação em domínio público dos detalhes da linguagem.

Specification Language for Encoding and Decoding (SLED) [RAM 97] é uma linguagem usada para descrição de instruções de máquina através de símbolos. O pacote de ferramentas *New Jersey Machine-Code Toolkit* (NJMCT) [RAM 94] corresponde a uma implementação SLED utilizada por aplicativos desmontadores e depuradores que manipulam código de máquina. Entretanto, seu depurador redirecionável, denominado `ldb` [RAM 92], apresenta algumas limitações como, por exemplo, a impossibilidade de avaliação de expressões que incluam chamadas de procedimentos.

ArchC [RIG 04] é uma ADL que está sendo desenvolvida no Laboratório de Sistemas da Computação (LSC) do IC-UNICAMP. Atualmente, ArchC permite a geração automática de simuladores, montadores e ligadores de código para diversas arquiteturas, tais como MIPS-I, SPARC-V8, Intel 8051, PowerPC e PIC 16F84. Adicionalmente, outras arquiteturas encontram-se em fase de especificação na ADL [ARC 07].

Dentre as vantagens de ArchC, destaca-se o fato de ser uma ADL de domínio público, que permite a inserção do modelo de simulação na plataforma, e a geração de simuladores em SystemC [SYS 07]. É possível a depuração do programa no simulador gerado para uma determinada arquitetura-alvo com o uso de um depurador redirecionado para a mesma arquitetura [RIG 04]. Contudo este processo pressupõe que já se possua o executável do depurador para a arquitetura-alvo. Um dos objetivos deste trabalho é justamente a geração automática do depurador e do desmontador (necessário ao depurador) a partir da descrição da arquitetura na ADL ArchC, cobrindo assim a lacuna existente na

geração de ferramentas binárias baseadas em ArchC.

A Figura 2.1, extraída de Baldassin [BAL 05], exibe a estrutura de uma descrição em ArchC.

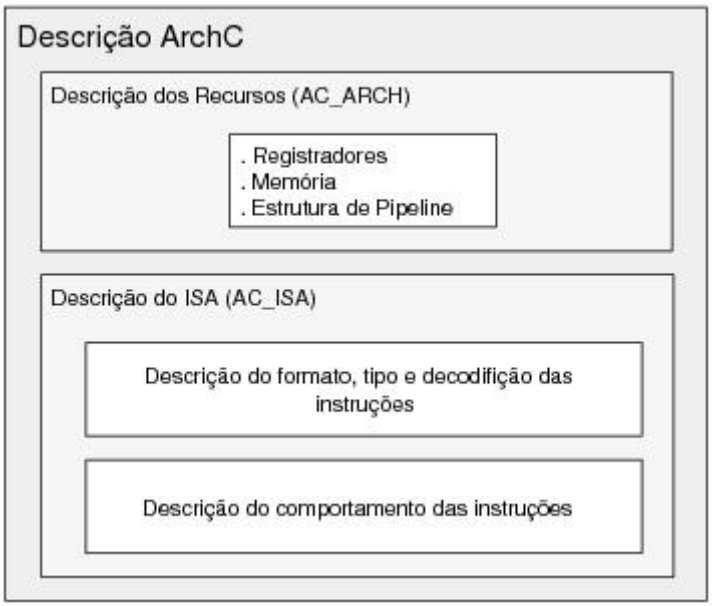

Figura 2.1: Estrutura de uma descrição em ArchC

A título de estudo de caso para compreensão dos recursos da ADL ArchC, foi desenvolvido um modelo funcional para o conjunto de instruções do modo THUMB do processador embarcado mais popular do mercado, a CPU ARM [PAT 04]. O modelo do THUMB mostrou-se robusto para uma variedade de experimentos a que foi submetido [KUS 06] conforme relatado no Apêndice B.

O Apêndice B.2 apresenta a descrição funcional de uma arquitetura na ADL ArchC, através do exemplo da modelagem do modo THUMB do processador ARM.

2.2 Ferramentas de geração e manipulação de código binário

Existe uma cadeia de ferramentas GNU utilizada para a construção de software, composta pelos pacotes *GNU Compiler Collection* (`gcc`) [STA 04a] e *GNU Binutils*

[PES 93], sendo este último voltado para a manipulação de arquivos-objeto, através de ferramentas como um montador (`gas`), um ligador (`ld`) e um desmontador (`objdump`) entre outras, além do *GNU Debugger* (`gdb`) [STA 04b].

No contexto deste trabalho, o pacote *GNU Compiler Collection* não é utilizado, enquanto que os pacotes *GNU Binutils* e *GNU Debugger* são amplamente utilizados.

2.2.1 GNU Binutils

O *GNU Binutils* está estruturado basicamente em um módulo *core*, independente de arquitetura, e em alguns módulos dependentes de arquitetura que devem ser reescritos para cada nova CPU-alvo. Esses módulos estão distribuídos em torno de duas bibliotecas principais:

- *Binary File Descriptor Library* (BFD) [CHA 91] - Fornece uma interface comum para todas as ferramentas (`gas`, `ld`, `objdump`), suportando diferentes formatos de arquivos-objeto.
- Opcodes [PES 93] - Possui informações sobre o conjunto de instruções da arquitetura (*Instruction Set Architecture* - ISA), tais como a forma de codificação e decodificação das instruções.

O `objdump` é o desmontador do pacote *GNU Binutils*, sendo utilizado principalmente como uma ferramenta para validação de arquivos objetos. Esta ferramenta permite que a partir de um arquivo-objeto se obtenha o seu respectivo código *assembly*, auxiliando assim os desenvolvedores no processo de detecção de erros. Através do uso das bibliotecas BFD e Opcodes e com a reescrita de um conjunto de rotinas dependentes de arquitetura, torna-se possível efetuar o redirecionamento do `objdump` para outra arquitetura-alvo.

A Figura 2.2 (a) exibe uma estrutura simplificada do funcionamento do `objdump`. Os módulos marcados com o símbolo '*' possuem componentes que precisam ser reescritos no caso de um novo redirecionamento. O *core* do desmontador oferece algumas funcionalidades, tais como a leitura do arquivo-objeto e o tratamento e exibição do nome

de seções. No redirecionamento para uma nova arquitetura devem ser implementadas rotinas pré-definidas pelo *core* do desmontador, as quais serão invocadas durante a geração da sintaxe *assembly* de um arquivo-objeto.

O desmontador `objdump` já foi redirecionado para vários processadores RISC e CISC e suporta os formatos de arquivo ELF, a.out e COFF.

O pacote *GNU Binutils*, em especial o `objdump`, possui características que viabilizam o seu redirecionamento para outras arquiteturas, justificando o seu uso no projeto ArchC, conforme será explicado na Seção 2.5.

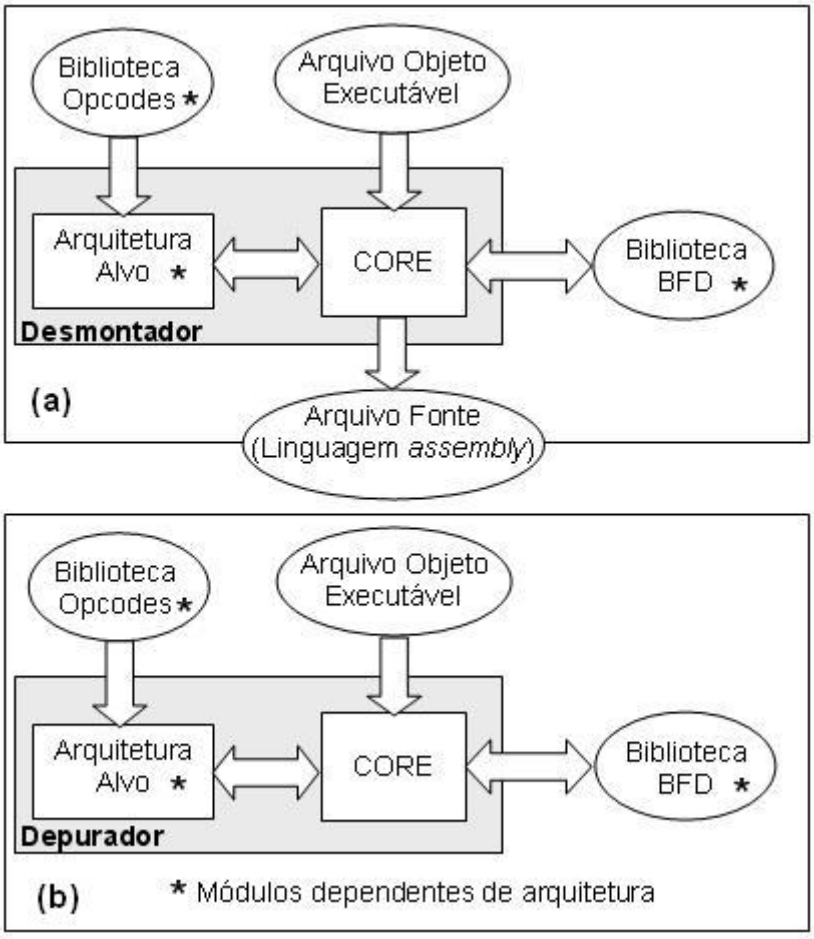

Figura 2.2: Estrutura das ferramentas GNU

2.2.1.1 Geração de ferramentas baseadas no pacote GNU Binutils

O trabalho de Abbaspour et al [ABB 02] trata de uma técnica de redirecionamento do pacote *GNU Binutils* para geração do montador `gas` e do ligador `ld` baseado em uma descrição abstrata do conjunto de instruções de uma arquitetura. Foram especificados e testados modelos para o processador SPARC e para um Intel 386. A partir dos modelos, são gerados os módulos dependentes de arquitetura para as bibliotecas BFD, Opcodes e os executáveis do montador e do ligador.

Neste trabalho foram identificados, também, os arquivos que necessitavam ser reescritos para o redirecionamento do desmontador mas não chegaram a ser implementados (somente o montador `gas` e o ligador `ld` o foram). A título de trabalho futuro os autores mencionam sua intenção de utilizar a mesma técnica para redirecionar o *GNU Debugger* (`gdb`).

2.2.2 GNU Debugger

A ferramenta *GNU Debugger* (`gdb`) [GDB 07] inicialmente fazia parte do pacote *GNU Binutils* e, com o passar do tempo, acabou sendo desmembrada e seguindo vida própria; contudo, a estrutura das bibliotecas do pacote continuam sendo um espelho do pacote *GNU Binutils*, sendo portanto as mesmas discutidas na Seção 2.2.1 e possibilitando assim a abordagem unificada das ferramentas.

O redirecionamento do `gdb` basicamente depende de algumas rotinas utilizadas pelo `objdump` (as quais fazem uso das bibliotecas BFD e Opcodes) e da reescrita de um módulo do depurador responsável pelo tratamento de *breakpoints*, chamada de procedimentos, definições de registradores que possuam um comportamento específico como o *frame pointer* (fp), *stack pointer* (sp), *return address pointer* (ra) e *instruction pointer* (pc).

A estrutura simplificada do funcionamento do `gdb` é mostrada na Figura 2.2 (b).

2.3 Ferramentas de otimização pós-compilação

É reconhecido que abordagens de otimização ao nível de linguagem *assembly*, como SALTO [SAL 07] e PROPAN [KäS 00], merecem ser investigadas [LEU 01]. Tais técnicas permitem que a infraestrutura convencional do compilador seja utilizada para habilitar otimizações de pós-compilação dependentes de máquina, a fim de prover maior qualidade ao código.

Embora otimizações pós-compilação sejam promissoras, elas podem inadvertidamente introduzir erros. Portanto, as ferramentas de inspeção de código não devem ficar restritas ao código-fonte, em vista de possíveis reordenamentos de instruções e diferentes usos de registradores após as otimizações. Conseqüentemente, os depuradores convencionais provavelmente negligenciarão os erros introduzidos por otimizações pós-compilação. Se a técnica utilizada para a geração do depurador exige que o mesmo seja executado sobre um arquivo-objeto, este tipo de erro poderá ser detectado.

2.4 Tradução binária

Tradução binária corresponde à conversão de um código objeto compilado para uma determinada arquitetura em um código de outra arquitetura-alvo, seja de forma estática ou dinâmica. Altman et al [ALT 00] abordam os diferentes tipos de tradução binária:

- Emulação - consiste na transcrição de instruções de um processador-alvo para o processador no qual o programa está rodando, em tempo de execução.
- Tradução dinâmica - corresponde a traduzir apenas partes do código do programa para outra arquitetura, ou seja, tradução de determinados blocos básicos.
- Tradução estática - é a tradução feita *offline*, possibilitando otimizações mais rigorosas no código do que na tradução dinâmica.

A emulação e a tradução dinâmica geram um acréscimo de tempo de execução. Já a tradução estática corresponde a uma ferramenta *stand-alone* que requer envolvimento

do usuário final. Como geralmente não se tem o código-fonte e somente o binário final do programa, sem as definições semânticas, não é possível fazer muitas otimizações no código traduzido como as efetuadas por um compilador [ALT 00].

No escopo de tradução estática, um trabalho interessante é FX!32 [CHE 97], que efetua a tradução binária de aplicações de 32 *bits* executadas em um microprocessador Intel x86 para um microprocessador Alpha. Um outro trabalho que merece destaque é o de Cifuentes et al [CIF 02], que aborda o *framework* de tradução binária UQBT, o qual é baseado em um *frontend* que traduz o código da arquitetura origem para uma representação intermediária passível de análise e em um *backend* que traduz da representação intermediária para código da arquitetura destino. Foram realizados experimentos com as CPUs SPARC, Pentium, MC68328, PA-RISC e ARM.

Os trabalhos encontrados que têm relação com tradução binária estática relatam dificuldade em se realizar tal implementação em vista das características específicas de cada arquitetura. O fluxo de tradução binária proposto no Capítulo 5 visa cobrir várias destas lacunas através da modelagem formal do comportamento das instruções com mapeamento para operações básicas.

2.5 A proposta deste trabalho frente aos trabalhos correlatos

As ferramentas propostas neste trabalho são um gerador de depuradores e um gerador de desmontadores baseados na especificação de processadores via ADL, sendo ferramentas de código livre, sem fins comerciais, em oposição a alguns trabalhos correlatos. A ADL escolhida foi ArchC, pois além de ser de domínio público, também apresenta os recursos para a especificação das características da arquitetura necessárias à geração das ferramentas de inspeção de código.

A técnica proposta para geração das ferramentas também é relevante em função das promissoras otimizações pós-compilação ao nível de *assembly* e pela necessidade contemporânea de ferramentas de depuração ao nível de sistema em plataformas heterogêneas.

Também corresponde a uma abordagem pragmática para redirecionamento automático de ferramentas, sendo o seu mecanismo de funcionamento claramente descrito e documentado, em oposição a alguns trabalhos correlatos.

Fazendo o redirecionamento automático dos pacotes *GNU Binutils* e *GNU Debugger*, além do ganho de produtividade na especificação do processador na ADL em relação ao redirecionamento manual (menor número de linhas de código), tem-se acesso a todos os recursos já implementados e bastante utilizados na execução das ferramentas de desmontagem e depuração de código, garantindo assim robustez e corretude de implementação. Além disso, as ferramentas geradas foram validadas tanto para processadores RISC como CISC, gerando evidências da generalidade das mesmas.

Capítulo 3

Estrutura e funcionamento das ferramentas

Este capítulo apresenta os algoritmos, em pseudocódigo, e descreve a lógica de execução das principais rotinas relacionadas à execução das ferramentas de desmontagem e depuração de código. A Seção 4.5 apresenta a formalização dos elementos de um processador utilizados nas ferramentas de desmontagem e depuração de código, sendo um insumo para uma implementação de geração destas ferramentas de maneira independente de ADL.

3.1 Lógica de execução das ferramentas

A fim de inspecionar o código binário e controlar sua execução, um depurador essencialmente requer uma `register-view` para o processador-alvo e uma rotina `instruction-tracking`.

A `register-view` provê o número de registradores de propósito geral endereçáveis (`gpr`), o *instruction pointer* (`pc`) e os ponteiros envolvidos na convenção de chamada de procedimentos: o *stack pointer* (`sp`), *frame pointer* (`fp`) e o *return address pointer* (`ra`). A Figura 3.1 mostra os elementos de uma `register-view` que devem estar referenciados no modelo do processador especificado na ADL.

```
register-view = (gpr, pc, sp, fp, ra)
```

Figura 3.1: Tupla de configuração dos registradores para o `gdb`

A rotina `instruction-tracking` é invocada sob demanda pelo depurador, obedecendo a configuração de *breakpoints* e *watch points* especificada no código. Esta rotina tem como função ler da memória o número de *bytes* necessários para identificar a instrução; essa identificação está presente na especificação dos formatos de instruções do processador.

A fim de prover o redirecionamento, esta rotina deve estar apta a trabalhar com formatos de instrução com tamanho fixo e variável. Uma vez que o tamanho da instrução é determinado, o desmontador é invocado. Como se deseja um redirecionamento automático para o depurador, a rotina de desmontagem também deve ser automaticamente redirecionada. Para ajudar na busca do tamanho da instrução é usada a rotina `get_lengths`, Algoritmo 1, a qual determina o conjunto de possíveis tamanhos de instrução para uma determinada arquitetura, onde `type-def` indica um formato de instrução do processador e `format-desc` a sua respectiva codificação binária dividida em campos.

Algoritmo 1 Função `get_lengths` - Extração do tamanho da instrução

1: $L \leftarrow \emptyset$;
2: **for** cada <*type-def*> **do**
3: $\quad\lambda \leftarrow$ tamanho de <*format-desc*>;
4: $\quad$**if** ($\lambda \notin L$) **then**
5: $\quad\quad L \leftarrow L \cup \{ \lambda \}$;
6: $\quad$**end if**
7: **end for**
8: **return** L;

O Algoritmo 2 descreve a rotina `get_asm`, responsável por obter a instrução a partir da representação binária lida. Seu funcionamento pode ser assim descrito: primeiramente todos os tamanhos de formatos de instrução vindos por parâmetro são armazenados em uma lista auxiliar na linha 1. O laço da linha 3, inicialmente assume que

a instrução corrente possui o menor tamanho de formato de instrução do modelo; em seguida, na linha 5, lê-se a instrução binária da memória. Na linha 6 é invocada a função `search_instruction` descrita no Algoritmo 3, que tem por função encontrar a instrução na `table-entry`, que corresponde a todas as instruções do processador obtidas do modelo na ADL. Caso encontrada uma instrução equivalente, a mesma é armazenada na linha 7 e na seqüência, linha 9, o tamanho do `instruction-format` é descartado e são procurados por outros formatos de instrução, do menor para o maior, a partir do mesmo endereço de memória. Por fim, na linha 11, para a instrução do último `instruction-format` encontrado, a rotina `disassemble` é invocada e o seu formato *assembly* é retornado. Caso não seja encontrada uma instrução equivalente para nenhum `instruction-format`, um erro será detectado. Justifica-se esta varredura para todos os tamanhos de formatos de instrução do modelo devido à existência de arquiteturas com tamanhos variáveis, onde não se sabe previamente qual o tamanho binário da instrução na memória a ser lido.

Algoritmo 2 Função `get_asm(pc, L)` - Recuperação da sintaxe *assembly*

```
1:  LAux ← L;
2:  J ← 0;
3:  while ( LAux not vazio ) do
4:      λ ← menor { LAux };
5:      I ← λ bytes da memória iniciando em pc;
6:      if search_instruction( I ) {Algoritmo 3} then
7:          J ← I;
8:      end if
9:      LAux ← LAux - { λ };
10: end while
11: if ( J > 0 ) then
12:     return disassemble( J ); {Algoritmo 4}
13: else
14:     return erro, instrução não encontrada no modelo;
15: end if
```

Algoritmo 3 Função `search_instruction(I)` - Busca da instrução

```
1: while ( table-entry not vazia ) do
2:     insn ← I & dmask;
3:     if ( insn = image ) and ( not pseudo-idx ) then
4:         return true; {instrução encontrada}
5:     end if
6: end while
7: return false; {instrução não encontrada}
```

O Algoritmo 4 descreve o processo de desmontagem do código binário de uma instrução. Primeiro, o mnemônico da instrução é decodificado na linha 1. Então, todos os operandos são verificados entre as linhas 2 e 5. Se um modificador estiver associado a um `operand type`, por exemplo, na codificação de desvios relativos ao `pc`, então o valor do respectivo campo da instrução é transformado de acordo com a função do modificador (no exemplo o endereço alvo da instrução é recuperado). Finalmente, o *assembly* correspondente à instrução é retornado na linha 6.

Algoritmo 4 Função `disassemble(instruction)` - Desmontagem da instrução

```
1: decodifica mnemonic-id da instrução;
2: for cada <oper-type> em <oper-type-list> do
3:     executa função <modifier> se existir;
4:     decodifica <oper-type>;
5: end for
6: return instrução em sintaxe assembly;
```

Algoritmo 5 Procedimento `configure` - Customização do depurador

```
1: L ← get_lengths(); {Algoritmo 1}
2: Associa a instruction-tracking como sendo a rotina get_asm( pc, L );
3: Extrai gpr, pc, sp, fp e ra de <operand-list> e <pointer-list>;
4: Register-view ← (gpr, pc, sp, fp, ra);
```

O Algoritmo 5 customiza o depurador através da rotina `instruction-tracking` e da `register-view`. Primeiramente, os tamanhos dos formatos de instrução são ob-

Algoritmo 6 Procedimento `execute_gdb` - Execução do depurador

1: Compila o programa com informações para o `gdb`;
2: Gera o simulador com suporte ao `gdb`;
3: Executa o programa no simulador;
4: Executa o `gdb` sob o mesmo programa;
5: Conecta o `gdb` na mesma porta de comunicação utilizada pelo simulador
6: Inicia o processo de depuração do código (`step`, `next`, `breakpoints`, `watchpoints`, etc)

tidos do modelo na linha 1; na linha 2 associa-se a rotina `instruction-tracking` como sendo a rotina `get_asm` e então são obtidos do modelo os números dos registradores pertinentes à configuração da `register-view`.

Com o depurador customizado para a arquitetura-alvo, as rotinas que efetuam o tratamento das funções de inspeção de código, como navegação no código-fonte, exibição do conteúdo de registradores e variáveis, que estão implementadas no *core* do `gdb`, são invocadas sob demanda, em função da necessidade do usuário através da especificação de comandos no *prompt* do `gdb`. O Algoritmo 6 estabelece o fluxo de execução do depurador em conjunto como o simulador da mesma arquitetura.

Capítulo 4

Geração automática de ferramentas redirecionáveis

Este capítulo apresenta as técnicas utilizadas para a geração automática de ferramentas redirecionáveis de inspeção de código desenvolvidas com base na ADL ArchC [ARC 07] e também uma formalização na Seção 4.5 como subsídio para geração das ferramentas de maneira independente de ADL.

A proposição destas técnicas contou com a colaboração dos acadêmicos Alexandre Mendonça e Felipe Carvalho, através de seus trabalhos de conclusão de curso [MEN 06a]. As Seções 4.1 e 4.2 tratam do processo de geração de ferramentas baseadas em ArchC e a Seção 4.3 descreve os arquivos gerados. Por fim, na Seção 4.4 são apresentados os resultados experimentais obtidos a partir do fluxo de validação dos geradores e das ferramentas geradas.

4.1 A geração de ferramentas baseadas em ArchC

ArchC possui um módulo denominado *ArchC pre-processor* (`acpp`) que é o responsável pela leitura dos modelos de arquitetura e criação de uma representação intermediária mantida em memória e portanto, reconstruída toda vez que uma ferramenta geradora é gerada [BAL 05].

A Figura 4.1, adaptada de [BAL 05], exibe a estrutura utilizada na geração de ferramentas baseadas em ArchC. É utilizada uma representação intermediária criada pelo módulo `acpp`, fornecendo assim insumos para as ferramentas geradoras.

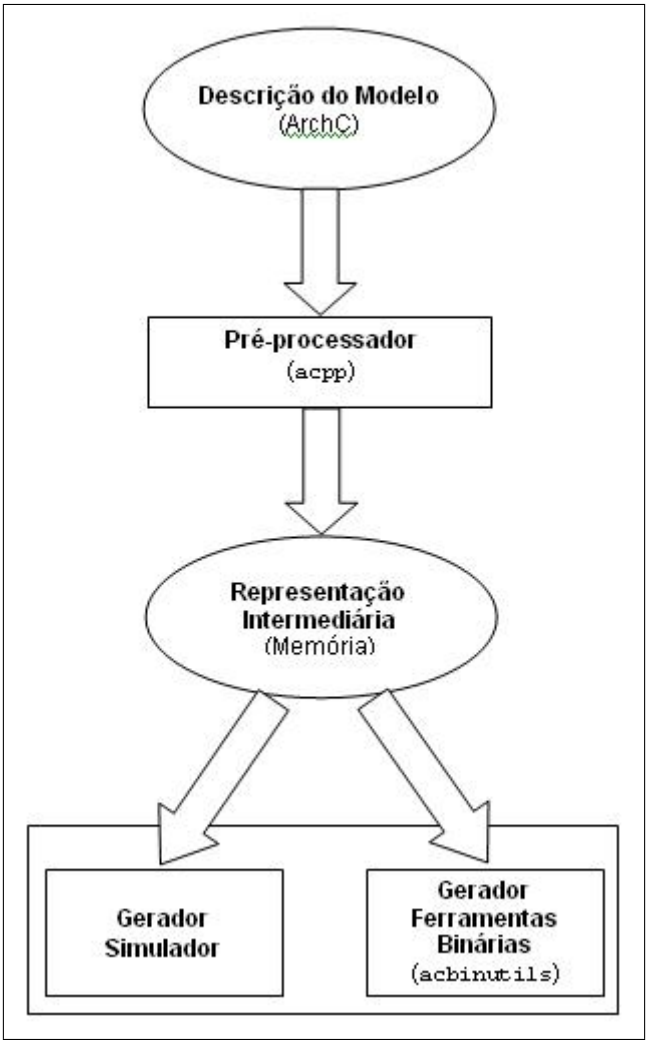

Figura 4.1: Estrutura para geração de ferramentas em ArchC

4.2 A geração de ferramentas binárias

A Figura 4.2, adaptada de [BAL 05], exibe o fluxo de geração dos executáveis das ferramentas binárias baseadas em ArchC, no qual foi incorporada a geração de desmontadores e depuradores de código. A partir das informações capturadas e armazenadas

pelo pré-processador, a ferramenta geradora de utilitários binários `acbinutils` gera e configura o conjunto de arquivos dependentes de arquitetura, os quais são inseridos nos pacotes *GNU Binutils* e *GNU Debugger*, para gerar o desmontador e o depurador respectivamente. Na seqüência, juntamente com os arquivos independentes de arquitetura, é feita a compilação dos pacotes, gerando-se os executáveis das ferramentas binárias da arquitetura em questão. O Apêndice A, mais especificamente a Seção A.1.2, apresenta um exemplo detalhado de instalação e uso da ADL ArchC para a geração de ferramentas binárias com o `acbinutils`.

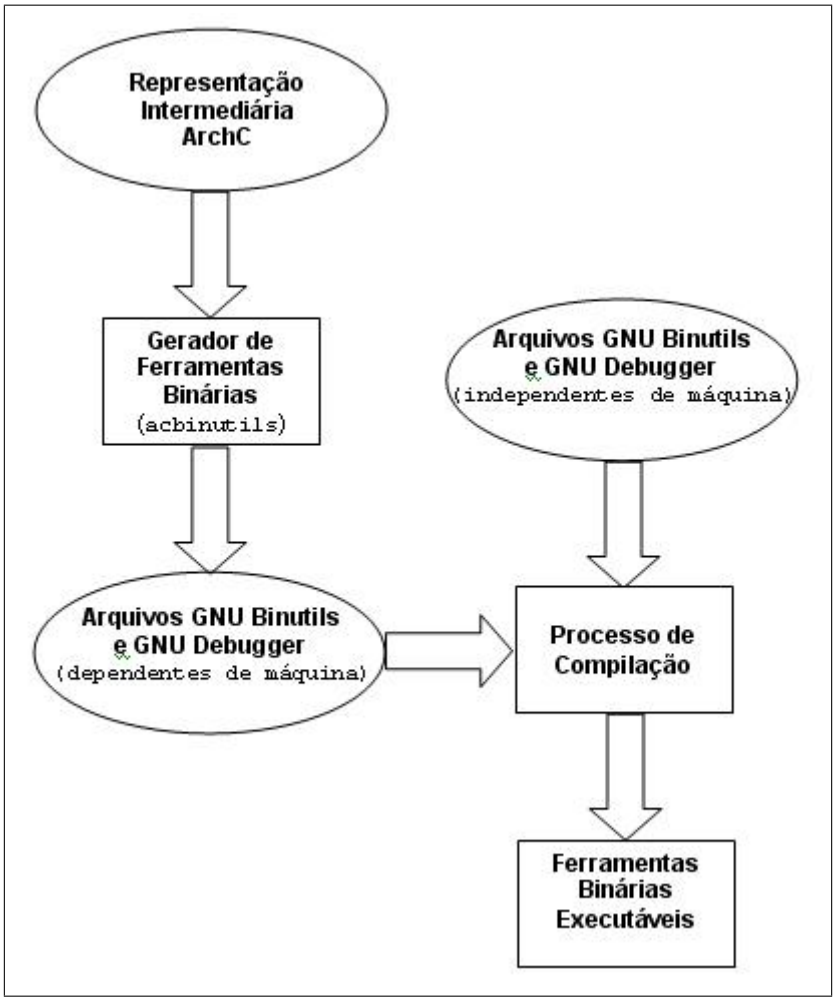

Figura 4.2: Fluxo de geração automática de ferramentas binárias executáveis no ambiente ArchC

4.3 Geração de arquivos para os pacotes GNU Binutils e GNU Debugger

```
- binutils              // GNU Binutils
  - bfd                 // biblioteca BFD
    . cpu-[arq].c
    . elf32-[arq].c
  - opcodes             // biblioteca Opcodes
    . [arq]-opc.c
    . [arq]-dis.c
  - include             // arquivos gerais de inclusão do pacote
    - elf
      . [arq].h
    - opcode
      . [arq].h
- gdb                   // GNU Debugger
  - bfd                 // biblioteca BFD
    . cpu-[arq].c
    . elf32-[arq].c
  - opcodes             // biblioteca Opcodes
    . [arq]-opc.c
    . [arq]-dis.c
  - include             // arquivos gerais de inclusão do pacote
    - elf
      . [arq].h
    - opcode
      . [arq].h
  - gdb                 // arquivos relacionados ao depurador
    . [arq]-tdep.c
    - config
      - [arq]
        . [arq].mt
```

Figura 4.3: Arquivos gerados no GNU Binutils e GNU Debugger para a arquitetura `[arq]`

Nas Seções 2.2.1 e 2.2.2 foram introduzidos os componentes básicos relacionados aos pacotes *GNU Binutils* e *GNU Debugger* respectivamente. Nesta seção são abordadas as bibliotecas e a técnica de redirecionamento para geração das ferramentas.

A estrutura de diretórios do pacote *GNU Binutils* (versão 2.16.1) e do pacote *GNU Debugger* (versão 6.4) relativa aos módulos utilizados neste trabalho é apresentada na Fi-

gura 4.3. O *GNU Debugger* fazia parte do *GNU Binutils*, sendo depois desmembrado. Por isso suas bibliotecas são um espelho das bibliotecas do *GNU Binutils*, acrescida apenas com a árvore de diretórios `gdb`, justificando assim a abordagem unificada das ferramentas, no que diz respeito ao funcionamento das bibliotecas. Os arquivos gerados na estrutura de diretórios dos pacotes *GNU Binutils* e *GNU Debugger* estão descritos na Figura 4.3.

Ao longo das próximas Subseções, `[arq]` representa uma nova arquitetura para a qual as ferramentas serão redirecionadas.

4.3.1 Biblioteca BFD

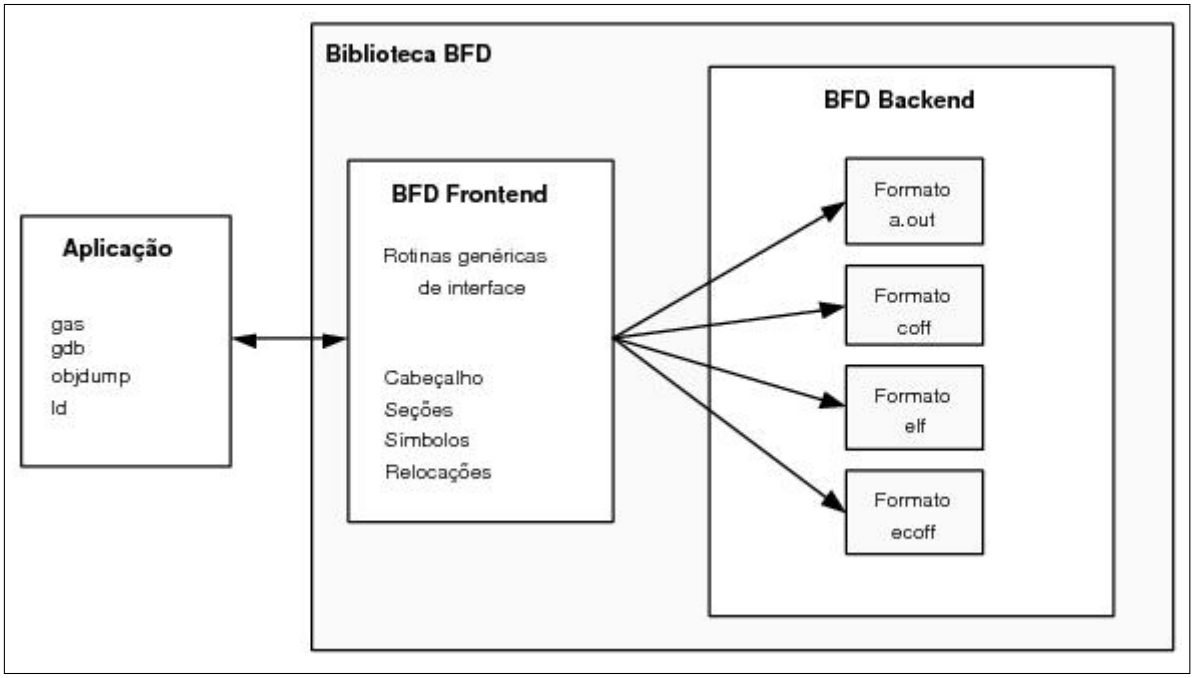

Figura 4.4: Organização da biblioteca BFD

A biblioteca BFD contém as rotinas para leitura e escrita de arquivos-objeto, ou seja, de uma forma genérica independente de um formato específico [BAL 05]. As ferramentas `objdump` e `gdb` utilizam as rotinas relacionadas à leitura.

A Figura 4.4, retirada de [BAL 05], exibe as duas partes que dividem conceitualmente a biblioteca:

- *frontend*: corresponde à interface da biblioteca com as aplicações e possui estruturas de dados canônicas para representar um arquivo-objeto. Define também qual o formato do *backend* utilizado e quando as rotinas deste formato devem ser executadas.

- *backend*: onde estão implementados os formatos específicos de arquivos objetos. Um *backend* deve prover rotinas que transformem suas estruturas de dados em estruturas canônicas utilizadas pelo *frontend*. Um novo formato de arquivo-objeto é suportado pela biblioteca através da criação de um novo *backend* contendo os arquivos genéricos de definição de formato e os arquivos específicos da arquitetura.

Os atributos do conjunto de instruções são extraídos do modelo do processador e codificados na biblioteca BFD. Para o formato ELF, que já existe na BFD e que é reconhecido pelo simulador gerado baseado no ambiente ArchC, são gerados os arquivos abaixo para cada nova arquitetura `[arq]`, onde `[pacote]` corresponde ao diretório do *GNU Binutils* ou do *GNU Debugger*:

- `[pacote]/bfd/elf32-[arq].c`: fornece ao *core* informações como o nome do formato, tamanho máximo de página e a função responsável pelo tratamento das relocações, entre outras.

- `[pacote]/bfd/cpu-[arq].c`: possui as informações referentes aos processadores de uma determinada arquitetura. Armazena informações como o número de *bits* da palavra, número de *bits* em um *byte* e o nome do processador, entre outras.

- `[pacote]/include/elf/[arq].h`: armazena a estrutura de dados do formato ELF utilizada pelos demais arquivos.

4.3.2 Biblioteca Opcodes

O conjunto de instruções de uma determinada arquitetura, assim como informações sobre codificação e decodificação de cada instrução, codificados em linguagem de programação `C`, fazem parte da biblioteca Opcodes.

Não existe uma estrutura comum a todas as arquiteturas para representação destas informações, sendo que o maior tempo de implementação gasto no porte manual de uma nova arquitetura concentra-se nesta biblioteca, justificando-se assim a automatização deste processo.

O `objdump` faz uso desta biblioteca para decodificar as instruções da arquitetura-alvo, identificando a partir da sintaxe *assembly* o estilo dos mnemônicos, nomes dos registradores, separação entre operandos, etc. O `gdb` invoca indiretamente esta biblioteca através do `objdump`, quando precisa exibir a sintaxe *assembly* de uma instrução para o usuário.

Para a especificação do conjunto de instruções de uma nova arquitetura `[arq]`, são gerados os arquivos abaixo, baseados na descrição do processador na ADL, onde `[pacote]` é o diretório do *GNU Binutils* ou do *GNU Debugger*:

- `[pacote]/opcodes/[arq]-opc.c`: onde fica especificado o conjunto de instruções da arquitetura.
- `[pacote]/opcodes/[arq]-dis.c`: armazena um conjunto de chamadas de funções escritas em linguagem `C`, geradas automaticamente a partir da descrição do processador na ADL, as quais serão usadas para decodificação das instruções da arquitetura. Corresponde ao principal arquivo relacionado ao desmontador.
- `[pacote]/include/opcode/[arq].h`: contém as estruturas e definições utilizadas pela biblioteca Opcodes.

As informações desta biblioteca estão armazenadas em três estruturas de dados desenvolvidas inicialmente por Baldassin [BAL 05], com acréscimo do campo `dmask` no contexto deste trabalho:

1. Opcodes: cada registro desta tabela corresponde a uma instrução especificada no modelo ArchC.
2. Símbolos: esta estrutura contém o mapeamento dos registradores especificados no modelo ArchC.

3. Pseudo-instruções: contém a lista de instruções codificadas para uma determinada pseudo-instrução.

A tabela de Opcodes é dividida em campos, conforme visualizado na Figura 4.5, onde cada campo tem o seguinte significado:

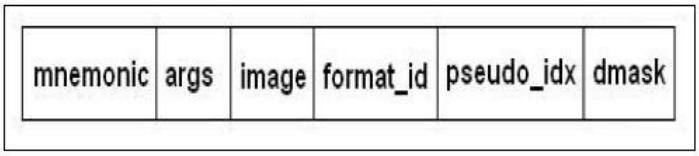

Figura 4.5: Estrutura de armazenamento da biblioteca Opcodes

- **mnemonic**: corresponde ao mnemônico da instrução.
- **args**: contém um índice para uma estrutura auxiliar que contém os argumentos da instrução, a saber: formatadores, posição do operando na instrução e informações relacionadas à relocação, necessária à geração de *link-editors*.
- **image**: armazena a imagem binária da instrução, a qual representa a parte fixa do código binário da instrução em questão. Para uma pseudo-instrução, este campo é anulado.
- **format_id**: corresponde ao identificador do formato da instrução. Durante o processo de decodificação, esse identificador é utilizado para obtenção do tamanho de cada operando da instrução.
- **pseudo_idx**: é um índice para a tabela de pseudo-instruções, a qual pode corresponder a uma ou mais instruções nativas. Esta informação é verificada pelo desmontador, pois o mesmo não exibe pseudo-instruções, mas apenas instruções nativas.
- **dmask**: este campo foi criado para viabilizar o desenvolvimento do gerador de desmontadores. Ele armazena a máscara utilizada para desmontagem das instruções. Esta máscara é gerada com base nos operandos que possuem conteúdo fixo na instrução, ou seja, esta máscara, juntamente com a imagem binária básica da instrução,

indicam ao desmontador quais *bits* são relevantes no processo de identificação da instrução.

A Figura 4.6 exibe parte de uma tabela de opcodes para a arquitetura MIPS-I.

```
{"add",   "%3:,%4:,%5:",  0x00000020,  0,  0,  0xFC00003F},
{"addi",  "%0:,%1:,%2:",  0x20000000,  1,  0,  0xFC000000},
{"b",     "%7:",          0x10000000,  1,  0,  0xFFFF0000},
{"beq",   "%1:,%0:,%7:",  0x10000000,  1,  0,  0xFC000000},
{"j",     "%4:",          0x00000008,  0,  0,  0xFC00003F},
{"jal",   "%9:",          0x0C000000,  2,  0,  0xFC000000},
{"lw",    "%0:,(%1:)",    0x8C000000,  1,  0,  0xFC00FFFF},
{"or",    "%0:,%1:,%6:",  0x34000000,  1,  0,  0xFC000000},
{"sw",    "%0:,%6:(%1:)", 0xAC000000,  1,  0,  0xFC000000},
{"subu",  "%0:,%0:,%6:",  0x00000000, 99, 36,  0x00000000}
```

Figura 4.6: Segmento de uma tabela de opcodes para o MIPS-I

A tabela de símbolos contendo os registradores da arquitetura é representada por uma tupla (símbolo, formatador, valor); um segmento desta tabela para o MIPS-I é ilustrado na Figura 4.7. Já a tabela de pseudo-instruções corresponde a uma lista de *strings* descrevendo as instruções que correspondem a uma pseudo-instrução [BAL 05], tal como exemplificado para a arquitetura MIPS-I, na Figura 4.8.

```
{"$0",    "reg",   0},
{"$1",    "reg",   1},
{"$zero", "reg",   0},
{"$at",   "reg",   1},
{"$kt0",  "reg",  26},
{"$kt1",  "reg",  27},
{"$gp",   "reg",  28},
{"$sp",   "reg",  29},
{"$fp",   "reg",  30},
{"$ra",   "reg",  31}
```

Figura 4.7: Segmento de uma tabela de símbolos para o MIPS-I

```
NULL,
"lui %0,\\%hi(%1)",
"addiu %0,%0,\\%lo(%1)",
NULL,
"lui %0,\\%hi(%1)",
"ori %0,%0,%1",
NULL,
"lui $at,%1",
"addu $at,$at,%2",
"sw %0,($at)",
NULL
```

Figura 4.8: Segmento de uma tabela de pseudo-instruções para o MIPS-I

4.3.3 O Desmontador `objdump`

O `objdump` é a ferramenta para desmontagem de código do pacote *GNU Binutils*. Ele faz uso da biblioteca BFD para leitura de arquivos objetos. As principais funcionalidades do `objdump` estão implementadas no seu *core*, onde também ficam definidas algumas rotinas que devem ser implementadas na parte dependente de arquitetura. A biblioteca Opcodes é utilizada pelo `objdump` para a decodificação das instruções e exibição de sua sintaxe *assembly* através das estruturas definidas na Seção 4.3.2.

Todos os arquivos relacionados ao desmontador ficam ou na biblioteca BFD ou na Opcodes, não existindo uma implementação em um arquivo à parte, como no caso do depurador `gdb`. A principal rotina do desmontador é a `print_insn_[arq]`, que fica na biblioteca Opcodes, no arquivo `[arq]-dis.c`. Esta rotina é invocada automaticamente pelo *core* do desmontador e corresponde ao principal trabalho de implementação quando do redirecionamento para uma nova arquitetura. A rotina `print_insn_[arq]` é responsável por ler um determinado número de *bytes* da memória, decodificar a instrução corrente, exibir a sua respectiva sintaxe *assembly* e retornar o número de *bytes* decodificados. A exibição de *labels*, da codificação binária da instrução e do endereço de memória da instrução são feitos automaticamente pelo *core* do desmontador.

4.3.4 O Depurador `gdb`

O `gdb` é a ferramenta para depuração de código do pacote *GNU Debugger*. Similar ao desmontador, o depurador também faz uso da biblioteca BFD para leitura de arquivos objetos da biblioteca Opcodes, para exibir, por exemplo, os nomes dos registradores. As principais funcionalidades do `gdb` são implementadas em seu *core*. Adicionalmente ao desmontador, o depurador exige a descrição de um módulo em linguagem `C` no arquivo `[arq]-tdep.c`, responsável pelo tratamento de *breakpoints*, *frames*, chamadas e retornos de procedimentos, etc.

O depurador invoca o desmontador para exibição da sintaxe assembly de uma instrução chamando a rotina `print_insn_[arq]` deste último. No primeiro passo da execução do depurador o *core* invoca a função `initialize_[arq]_tdep`, que indica quais funções deste arquivo são responsáveis por tratar alguns aspectos da arquitetura. Os principais métodos são listados na seqüência, indicando sua respectiva funcionalidade:

- `set_gdbarch_register_type`: recebe como parâmetro o número do registrador no banco de registradores e retorna o tipo do mesmo, como por exemplo: inteiro sinalizado de 32 *bits*, ponto flutuante de 32 *bits*, entre outros.

- `set_gdbarch_register_name`: recebe como parâmetro o número do registrador no banco de registradores e retorna o seu nome simbólico através da leitura da tupla na biblioteca Opcodes que armazena os registradores da arquitetura.

- `set_gdbarch_inner_than`: indica ao *core* a direção do crescimento da pilha, indicado pelos atributos `core_addr_lessthan` e `core_addr_greaterthan`.

- `set_gdbarch_breakpoint_from_pc`: indica ao *core* qual função faz o tratamento de *breakpoints*, que corresponde, de uma maneira geral, a um ponto de parada do programa definido pelo usuário, a partir do qual se dará a execução do programa. Esta função retorna o código `opcode` da instrução responsável pelo *breakpoint* da arquitetura.

- `set_gdbarch_print_insn`: indica ao *core* qual a função de desmontagem

que será utilizada para exibir as instruções em sintaxe *assembly* da arquitetura. A função indicada pertence ao módulo desmontador abordado na Seção 4.3.3.

- `[arq]_frame_cache`: de maneira simplificada, este método tem a função de criar *frames* para o tratamento de subrotinas; ele analisa o prólogo da subrotina para produzir um *backtrace* e permitir ao usuário a manipulação de variáveis e argumentos de *frames* antigos, pois o `gdb` necessita encontrar o endereço-base de *frames* antigos e descobrir onde os registradores deste *frame* foram salvos. Também guarda o endereço de retorno do *frame* e salva o contexto (valores dos registradores), além de outras funções auxiliares.

4.4 Resultados experimentais

Os experimentos foram executados em um computador com CPU Intel® Pentium 4 (3 GHz), com 1 GB de memória principal. O sistema operacional utilizado foi o Debian GNU/Linux.

Para validação das ferramentas foi utilizado o já conhecido pacote de *benchmarks* MiBench [GUT 01] para experimentos com as CPUs MIPS, PowerPC e SPARC. Para experimentos com a CPU i8051, um conjunto mais simples de *benchmarks* foi utilizado, o Dalton [DAL 07], pois o *benchmarks* MiBench requer a manipulação de arquivos, a qual não é uma funcionalidade apropriada para microcontroladores. Os geradores de montadores [BAL 05] e de ligadores [CAS 06] utilizados foram os já disponíveis na ferramenta `acbinutils`. Para a geração dos desmontadores e dos depuradores, foram utilizados os geradores desenvolvidos no contexto deste trabalho.

A fim de validar as ferramentas desenvolvidas, ferramentas convencionais, nativas, foram redirecionadas manualmente, visando a geração de arquivos e valores de referência, os quais foram comparados com os resultados obtidos a partir das ferramentas geradas.

Para validar as ferramentas geradas, além da verificação da adequação de suas funcionalidades, também foi observada a sua capacidade de redirecionamento. Para atestar esta última, o procedimento de validação descrito nas seções subseqüentes foi repetido

para quatro CPUs: PowerPC, MIPS, SPARC e i8051.

4.4.1 Fluxo de validação do desmontador

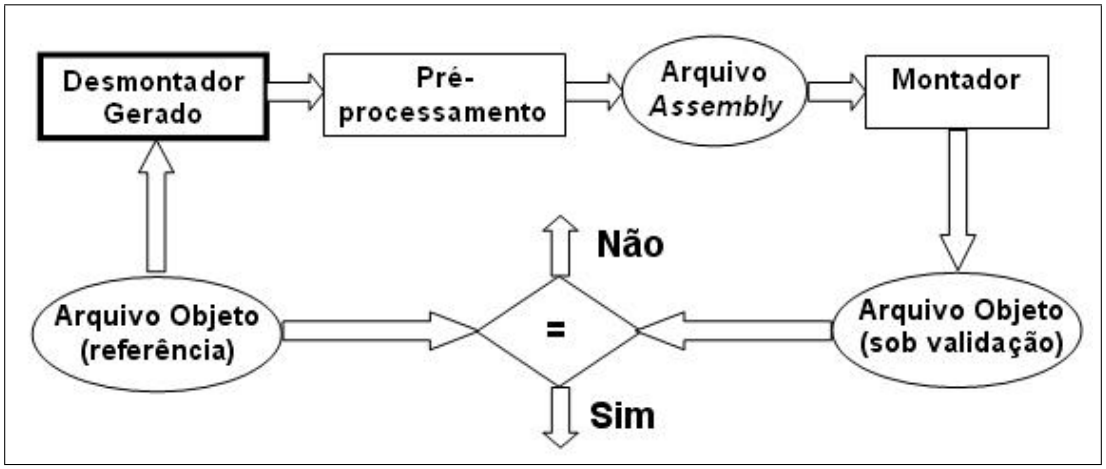

Figura 4.9: Fluxo de validação do desmontador

Para validar a ferramenta geradora de desmontadores, foi adotada a idéia que segue: dado um arquivo-objeto de referência (*benchmark*), se ele for desmontado e então remontado, o arquivo resultante deve ser igual ao arquivo de referência.

A Figura 4.9 mostra o fluxo de validação adotado. Retângulos representam ferramentas e elipses denotam arquivos. O procedimento inicia com um arquivo-objeto de referência sendo submetido ao desmontador gerado (a ser validado), que por sua vez gera o respectivo código em linguagem *assembly* do arquivo-objeto. Por padrão, o desmontador exibe o código *assembly* para verificação do arquivo-objeto. Para que se torne um *assembly* passível de ser remontado, é necessário realizar um pré-processamento que retira informações desnecessárias e adiciona *labels* para os desvios. Após o pré-processamento, o *assembly* resultante é submetido ao montador, dando origem a um novo arquivo-objeto. Para finalizar, o arquivo original de entrada (referência) e o arquivo de saída (sob validação) são comparados em suas seções *".text"*, pois é nesta seção que se encontra o código executável da aplicação, a fim de se verificar se são iguais ou não.

Alternativamente, a validação poderia ser feita através da comparação do código

assembly obtido a partir do código objeto e do código anterior original. Contudo, a comparação direta do *assembly* pode deparar-se com a presença de pseudo-instruções ou instruções que admitem mais de uma sintaxe *assembly*. Por exemplo, a instrução *"jump at register"* do MIPS pode ser escrita de duas formas diferentes: "`jr $1`" ou "`j $1`". Em função disto foi utilizado o procedimento de desmontagem e remontagem de código, sem perda da generalidade. O procedimento de validação foi repetido para cada CPU-alvo e para cada programa do *benchmark*. Como resultado, verificou-se a igualdade em todas as comparações realizadas, constatando-se assim a corretude do desmontador para os casos testados.

Também foram considerados aspectos relativos à eficiência das ferramentas, como a relação entre o tamanho do código do programa e o tempo para desmontagem pela ferramenta. Como a ferramenta de depuração invoca procedimentos de desmontagem, os resultados servem para constatar a eficiência de ambas as ferramentas.

Para verificar-se o potencial de redirecionamento da ferramenta geradora, o procedimento foi repetido para CPUs RISC (PowerPC, MIPS, SPARC) e CISC (i8051), cujos resultados são exibidos na Tabela 4.1 e na Tabela 4.2, respectivamente. As duas primeiras colunas mostram o programa *benchmark* e o seu respectivo número de arquivos. As demais colunas contêm o tamanho da seção *".text"* e os tempos de desmontagem para cada CPU-alvo distinta na ferramenta gerada (`acdsm`) e na nativa (`objdump`). Como não existe porte para o i8051 no pacote *GNU Binutils*, a coluna de tempo de desmontagem na ferramenta nativa foi suprimida na Tabela 4.1. Comparando o tempo obtido com a ferramenta gerada em relação ao desmontador nativo do *GNU Binutils* (`objdump`) na Tabela 4.2, 18 *benchmarks* apresentaram um melhor tempo, 8 levaram o mesmo tempo e 31 foram mais lentos. Na média, o tempo obtido com a ferramenta gerada foi 1,15 vezes mais lento que o desmontador nativo. Isto acaba sendo um preço a ser pago em função do benefício de a ferramenta ser redirecionável de maneira automática. Contudo já foram identificadas oportunidades para otimização das ferramentas e conseqüente redução dos seus tempos de execução.

Tabela 4.1: Resultados para processador CISC (i8051)

Programa	Arquivos	Tamanho da seção ".text"[b]	Tempo de execução [s]
cast	1	213	0.002
divmul	1	351	0.002
fib	1	442	0.003
gcd	1	186	0.002
int2bin	1	188	0.002
negcnt	1	173	0.002
sort	1	425	0.003
sqroot	1	1135	0.007
xram	1	214	0.003

Tabela 4.2: Resultados para processadores RISC

Programa	**Arquivos**	**Tamanho da seção ".text"[Kb]** **Tempo de execução [s] (`acdsm | objdump`)**					
		MIPS		**SPARC**		**PowerPC**	
basicmath_small	4	5.1		5.2		4.9	
		0.007	0.007	0.013	0.008	0.010	0.008
basicmath_large	4	6.3		6.3		5.9	
		0.012	0.009	0.013	0.012	0.012	0.010
bitcount	9	4.9		4.1		4.1	
		0.010	0.006	0.010	0.009	0.009	0.008
qsort_small	1	1.1		1.1		1.0	
		0.003	0.003	0.004	0.005	0.003	0.003
qsort_large	1	1.8		2.0		1.7	
		0.003	0.004	0.006	0.008	0.005	0.004
susan	1	64.7		59.4		52.7	
		0.099	0.074	0.139	0.057	0.104	0.095
jpeg	60	284.8		239.8		228.2	
		0.442	0.317	0.537	0.219	0.437	0.406
typeset	1	29.7		32.6		25.3	
		0.049	0.035	0.071	0.031	0.050	0.039
dijkstra_small	1	2.5		2.4		2.2	
		0.004	0.006	0.005	0.008	0.007	0.005
dijkstra_large	1	2.5		2.4		2.2	
		0.003	0.004	0.006	0.006	0.005	0.006
ispell	1	1.2		1.1		1.0	
		0.002	0.003	0.004	0.006	0.002	0.003
stringsearch_small	4	5.6		5.3		4.9	
		0.011	0.009	0.012	0.010	0.012	0.010
stringsearch_large	4	5.6		5.6		4.9	
		0.012	0.008	0.014	0.011	0.010	0.011
blowfish	7	19.2		16.4		15.7	
		0.036	0.028	0.041	0.020	0.036	0.029
sha	2	3.3		2.8		2.7	
		0.007	0.005	0.006	0.007	0.005	0.005
rawcaudio	2	2.5		2.4		2.1	
		0.006	0.005	0.006	0.006	0.005	0.005
rawdaudio	2	2.5		2.3		2.1	
		0.002	0.003	0.004	0.006	0.005	0.005
crc32	1	1.3		1.2		1.2	
		0.002	0.003	0.001	0.006	0.003	0.003
fft	3	5.9		5.5		5.3	
		0.010	0.007	0.014	0.010	0.012	0.012

4.4.2 Fluxo de validação do depurador

O depurador gerado comunica-se com o respectivo simulador da mesma arquitetura via porta serial, assumindo assim a execução do programa que está sendo simulado. Para que o simulador gerado com base na descrição da arquitetura em ArchC suporte a conexão do depurador, ele deve ser gerado com o parâmetro `-gdb`. Mais detalhes sobre a geração do simulador com suporte à depuração podem ser obtidos em [ARC 07].

O código executável do programa a ser submetido ao simulador com suporte à depuração também deve ter sido compilado de maneira que contenha as informações para depuração, parâmetro `-g` do *GNU Compiler Collection* (`gcc`) [STA 04a].

Para validar a ferramenta de depuração gerada, foi definido um conjunto de *breakpoints* e *watchpoints* para cada arquivo executável testado e foram observados os valores produzidos pelo depurador convencional (redirecionado manualmente no *GNU Debugger*) e pelo depurador gerado. Foram utilizados os *benchmarks* em três diferentes CPUs alvo: MIPS, PowerPc e SPARC. Não foi possível efetuar a validação do depurador gerado para a CPU i8051 pois o mesmo necessita que o arquivo-objeto submetido à simulação e conseqüente depuração tenha sido compilado por um *cross compiler* que gere informações para depuração, similar ao parâmetro `-g` do GCC [STA 04a]. Entretanto, não foi encontrado um *cross compiler* para o i8051 com tal característica.

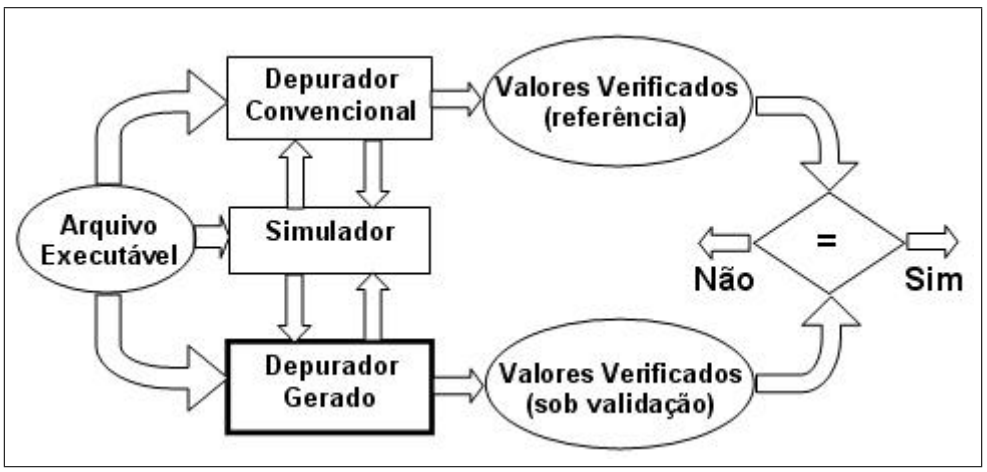

Figura 4.10: Fluxo de validação do depurador

A Figura 4.10 mostra o fluxo de validação utilizado. Retângulos representam ferramentas, enquanto elipses representam ou um arquivo ou um conjunto de valores observados. O procedimento envolve dois fluxos idênticos, primeiramente usando o simulador com o depurador convencional e na seqüência com o depurador gerado. O processo de validação considera um determinado arquivo executável, compilado para a arquitetura-alvo, sendo alimentado no simulador gerado com suporte à depuração. Primeiramente, um conjunto pré-definido de *breakpoints* e *watchpoints* são inseridos no código pelo depurador convencional. Como resultado tem-se um código instrumentado sendo executado. Então, os valores destes *watchpoints* são anotados como referência. Na seqüência o depurador gerado é usado, repetindo-se o mesmo procedimento, exatamente com os mesmos *breakpoints* e *watchpoints*. Por fim, os valores obtidos são comparados aos valores de referência. Em média 50 valores por *benchmark*, sendo este valores de variáveis locais, globais e de ponteiros que sofrem modificação tanto fora como dentro de procedimentos.

Este procedimento de validação foi repetido para cada CPU-alvo e para cada programa do *benchmark*. Depois de todas as comparações efetuadas, constatou-se que, com o uso dos depuradores gerados pela ferramenta de redirecionamento automático, os valores observados foram equivalentes aos obtidos usando-se os depuradores do pacote *GNU Debugger* redirecionados manualmente, levando assim a uma forte evidência da corretude do depurador gerado.

Quanto à técnica de validação utilizada, vale ressaltar que também foram exercitados os recursos de depuração de sub-rotinas (`step` e `next`) através da verificação *watchpoints* em variáveis locais de procedimentos e funções.

4.5 Formalização do conjunto de instruções de um processador

Esta seção descreve a formalização do conjunto de instruções de um processador através de uma Gramática Livre de Contexto (GLC) usando notação BNF [BAC 79], visando fornecer subsídios para a geração automática de ferramentas de inspeção de código

a partir de uma ADL arbitrária. Inicialmente é apresentado um exemplo para facilitar o entendimento da especificação realizada.

4.5.1 Formalização das propriedades específicas

A Figura 4.11 especifica a estruturação formal das informações tipicamente encontradas nos manuais dos processadores, abordando noções de instruções, pseudo-intruções, operandos e modificadores.

Um modificador é uma função que transforma o valor de um dado operando; essa transformação é necessária, por exemplo, em instruções de desvio relativo ao `pc` ou em instruções que manipulam os *bits* de mais alta ordem. A função modificadora possui quatro variáveis pré-definidas para especificar a transformação: `input` é o valor original do operando, `address` representa a localização da instrução, `parm` é um parâmetro que contém um valor auxiliar (como o número de *bits* para deslocamento), `output` retorna o valor do operando transformado.

```
<isa-def> ::= <operand-list> <pointer-list> <modifier-list>
  <type-list> <instruction-list> <pseudo-list>

<operand-list> ::= <operand-def> <operand-list> | <operand-def>

<operand-def> ::= operand oper-id { "mapping definition" }

<pointer-list> ::= stack-pointer sp-id { "mapping definition" }
  frame-pointer fp-id { "mapping definition" }
  instruction-pointer pc-id { "mapping definition" }
  return-address-pointer ra-id { "mapping definition" }

<modifier-list> ::= <modifier-def> <modifier-list> | empty

<modifier-def> ::= modifier modifier-id { "modifier code" }

<type-list> ::= <type-def> <type-list> | <type-def>

<type-def> ::= type type-id { <format-desc> }

<format-desc> ::= field-id : constant , <format-desc> | field-id : constant

<instruction-list> ::= <instruction-def> <instruction-list> | <instruction-def>

<instruction-def> ::= instruction insn-id { type-id ; (<syntax-desc>) :
  ( <operand-decoding> ) ; <opcode-decoding> }

<syntax-desc> ::= mnemonic-id <oper-type-list>

<oper-type-list> ::= <qualifier> <oper-type> , <oper-type-list> | <qualifier> <oper-type>

<oper-type> ::= oper-id | imm | addr <modifier> | exp <modifier>

<modifier> ::= << modifier-id ( constant ) | empty

<operand-decoding> ::= field-id , <operand-decoding> | field-id

<opcode-decoding> ::= field-id = constant , <opcode-decoding> | field-id = constant

<qualifier> ::= # | $ | empty

<pseudo-list> ::= pseudo_instr ( <syntax-desc> ) {
  <syntax-desc-pseudo>
}

<syntax-desc-pseudo>  ::= <syntax-desc> ; <syntax-desc-pseudo> | <syntax-desc> ;
```

Figura 4.11: Modelo abstrato de um processador

```
 1. operand reg { $[0..90] = [0..90];
 2.               $zero = 0;
 3.               $at = 1; }
 4.
 5. stack-pointer sp { $sp = 29; }
 6. frame-pointer fp { $fp = 30; }
 7. instruction-pointer pc { $pc = 37; }
 8. return-address-pointer ra { $ra = 31; }
 9.
10. modifier R { output = input + address + parm; }
11.
12. type Type_I { op:6, rs:5, rt:5, imm:16 }
13.
14. instruction beq {
15.   Type_I;
16.   (beq reg, reg, exp << R(2)) : (rs,  rt,  imm);
17.   op=0x04
18. }
19.
20. pseudo_instr( "ble %reg, %reg, %exp" ) {
21.   "slt $at, %1, %0";
22.   "beq $at, $zero, %2";
23. }
```

Figura 4.12: Segmento do modelo do MIPS-I

O tipo de operando `oper-type` especifica a natureza do campo da instrução e pode estar atrelado a um valor binário codificado para o campo em questão. Exemplos de tipos de operando são `imm`, para valores imediatos, `addr`, para endereços simbólicos e `exp`, para expressões envolvendo valores imediatos e símbolos.

A Figura 4.12 mostra um exemplo de parte da modelagem de um processador de acordo com a sintaxe especificada. A linha 1 descreve o mapeamento para o operando `reg`, onde os símbolos `$0`, `$1`, ..., `$90` são mapeados para os valores 0, 1, ..., 90. Observe-se que o mapeamento de muitos para um é possível. Por exemplo, os símbolos `$zero` e `$at` são mapeados para valores já mapeados na linha 1. Entre as linhas 5 e 8 estão definidos alguns registradores utilizados na chamada de sub-rotinas, `$sp`, `$fp` e `$ra`, além do registrador que aponta para a próxima instrução, o `$pc` . A linha 10 define o modificador `R`, que especifica uma função para ser aplicada para transformações relati-

vas ao `pc`. O resultado do modificador (`output`) é obtido através da soma da localização atual (`address`) com o valor do operando (`input`) e também a um *offset* (`parm`). A linha 12 define um formato de instrução da arquitetura através da lista dos campos com seus respectivos tamanhos em *bits*, no caso o formato tipo `Type_I`. As linhas de 14 a 18 definem a instrução `beq`. A linha 15 associa o formato `Type_I` a instrução. A linha 16 especifica a sintaxe *assembly*: `reg`, `reg` e `exp` são relacionados aos campo da instrução `rs`, `rt` e `imm` (`beq` é o mnemônico da instrução). O modificador `R` (com *offset* de 2) é aplicado ao tipo de operando `imm`, que por sua vez especifica que o valor resultante é relativo ao `pc` e deslocado 2 *bits* para a esquerda. Na linha 17, o valor constante 0x04 está associado ao campo `op` da instrução. Finalmente, entre as linhas 20 e 23 está especificada a pseudo-instrução `ble`, que é expandida para outras duas instruções nativas, `slt` (não especificada neste modelo) e `beq`.

Uma tabela gerada que contém as instruções do processador é o ponto de entrada para os algoritmos utilizados pelas ferramentas de desmontagem e depuração de código. Cada entrada nesta tabela é definida por uma tupla conforme a Figura 4.13:

```
table-entry = (mnemonic, args, image, format_id, pseudo_idx, dmask)
```

Figura 4.13: Tupla da tabela de instruções

A Figura 4.14 ilustra o significado destes elementos através de um exemplo com base no modelo da Figura 4.12 para a instrução `beq`:

```
{"beq", "%reg:1:,%reg:2:,%exp:3:", 0x10000000, Type_I, 0, 0xFC000000}
```

Figura 4.14: Exemplo da instrução `beq` na tabela de instruções

O primeiro elemento é o mnemônico da instrução (`beq`). O segundo armazena informações como o tipo (`reg`, `reg`, `exp`) e localização dos campos na instrução (`1`, `2`, `3`). O terceiro elemento armazena a imagem binária parcial da instrução (`0x10000000`), contendo o conteúdo dos campos que identificam a instrução. O quarto elemento identifica o formato da instrução (`Type_I`, neste caso). O quinto elemento especifica se a entrada na tabela de instruções se refere a uma pseudo-instrução ou não (`0` = não). Fi-

nalmente, o último elemento armazena a máscara (`0xFC000000`) para ser usada pelo algoritmo do desmontador, que indica quais campos devem ser observados no processo de identificação da instrução.

Capítulo 5

O papel das ferramentas de inspeção de código na tradução binária

Este capítulo propõe uma técnica para tradução binária, a qual é baseada na geração automática de ferramentas a partir de modelos de CPUs descritos através de uma ADL. A motivação, a proposta e o estudo experimental de viabilidade dessa técnica foram realizados cooperativamente pelo autor e dois outros mestrandos cujas dissertações abordam tópicos também relevantes para a tradução binária [CAS 07] [FIL 07]. Por essa razão, o texto das Seções 5.1, 5.2 e 5.4 é deliberadamente comum às três dissertações de mestrado. Entretanto, como a implementação do protótipo contou com contribuições distintas e complementares, a Seção 5.3 descreve a contribuição específica do autor para esse trabalho conjunto.

5.1 Motivação

Como discutido no Capítulo 1, durante a exploração do espaço de projeto de um SoC, pode-se ter que avaliar o impacto de várias CPUs alternativas até que os requisitos sejam satisfeitos. Isso requer a rápida geração de código executável para cada uma das CPUs exploradas, a qual pode ser obtida de três maneiras distintas:

- Alternativa 1 - Disponibilidade de compiladores convencionais portados para cada

uma das CPUs candidatas;

- Alternativa 2 - Disponibilidade de um compilador redirecionável;
- Alternativa 3 - Disponibilidade de um compilador convencional para uma das CPUs e de um tradutor binário capaz de gerar código executável para as demais CPUs.

A Alternativa 1 restringe o espaço de soluções exploradas ao uso de CPUs tradicionais, cujo uso intensivo justificou o desenvolvimento de compiladores próprios.

A Alternativa 2 é a mais genérica, mas requer o uso de compiladores redirecionáveis, os quais são invariavelmente proprietários (como é o caso do compilador LisaTek [COW 07]) e cujas licenças são caras, já que sua oferta no mercado é bastante pequena.

A Alternativa 3 tem a vantagem de requerer apenas um compilador convencional portado para uma única arquitetura, cuja licença pode ser pública (como a do `gcc`), desde que se disponha de um tradutor binário para redirecionar o código para as demais arquiteturas a serem exploradas.

Em princípio, um tradutor binário poderia ser obtido através do encadeamento de geradores de utilitários binários (Seção 5.2). Ora, a ADL ArchC provê geradores de utilitários binários sob licença GPL. Assim, o desenvolvimento de um tal tradutor resultaria numa solução de baixo custo para a exploração de CPUs. É de se esperar que o esforço de desenvolvimento de um tradutor binário estático - restrito às necessidades de sistemas embarcados - seja inferior ao requerido para se desenvolver um compilador redirecionável.

A principal dificuldade é garantir que a qualidade do código traduzido não seja muito inferior à obtida através de um compilador. Supondo que o compilador tenha realizado otimizações independentes de arquitetura antes de gerar o código executável sob tradução, é de se esperar que o código traduzido tenha qualidade similar, desde que o tradutor suporte otimizações dependentes de arquitetura utilizadas em compiladores-otimizadores contemporâneos, tais como seleção de instruções, escalonamento de código e alocação de registradores.

Na revisão da literatura realizada, não foi encontrado trabalho de pesquisa similar utilizando geração automática de ferramentas a partir de ADL para fins de tradução bi-

nária no contexto de exploração do espaço de projeto. Este fato, aliado à infra-estrutura disponibilizada pelo pacote ArchC, motivou a investigação da viabilidade dessa alternativa, como reportado nas próximas seções.

5.2 Proposta de estrutura de um tradutor binário

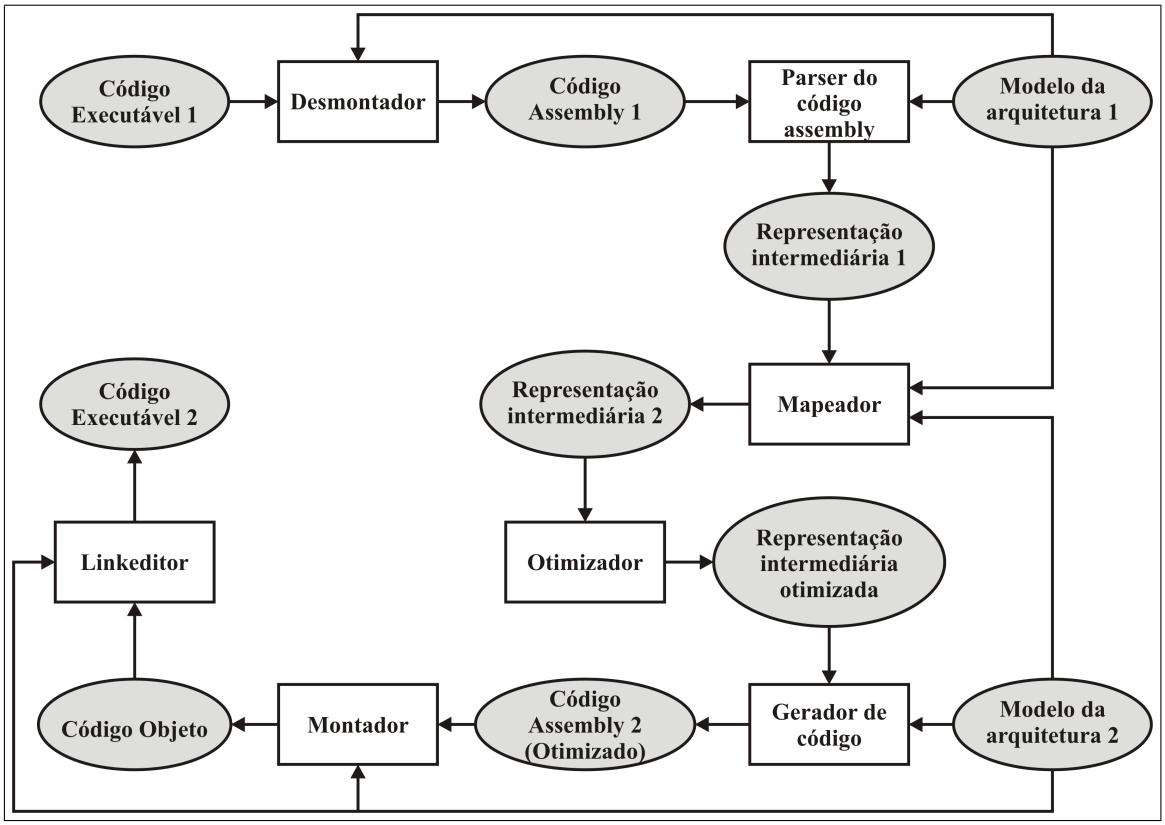

Figura 5.1: Fluxo de tradução binária

O tradutor binário proposto neste capítulo realiza a tradução estaticamente. Uma possível estrutura de um tradutor binário estático é ilustrada na Figura 5.1.

O código executável a ser traduzido deve inicialmente ser transformado em código *assembly* através de um desmontador. Em seguida, um *parser* do código *assembly* utiliza informações específicas da arquitetura para que seja gerada uma representação intermediária do código. Em geral, este tipo de representação intermediária deve ser uma representação em mais alto nível das instruções do código *assembly* e deve ser independente

da arquitetura-alvo.

A partir desta representação e de informações como sintaxe e semântica das instruções da arquitetura para a qual o código foi originalmente gerado, o mapeador deve operar fazendo com que ela seja traduzida em outra representação similar, buscando instruções da arquitetura-alvo que tenham semântica equivalente. É gerada então uma nova representação que corresponde às instruções da arquitetura-alvo que sejam equivalentes às instruções da arquitetura original.

A representação resultante pode ser o ponto de entrada para um otimizador que opera antes da geração do código. A partir desta representação traduzida e otimizada, usando como insumos as informações específicas da arquitetura-alvo, a ferramenta deve gerar o código *assembly* equivalente ao original, o qual pode ser então montado e linkeditado para dar origem ao código executável traduzido.

As ferramentas usadas nos passos intermediários (desmontador, montador e linkeditor) são geradas automaticamente a partir dos modelos das arquiteturas fonte e alvo descritos através de uma ADL.

5.3 A contribuição para o estudo de viabilidade

5.3.1 Exemplo do processo de tradução binária

A resolução do problema de mapeamento de instruções foi feita pelo mestrando Daniel C. Casarotto [CAS 07] e a otimização pelo mestrando José O. C. Filho [FIL 07]. O autor desta dissertação desenvolveu o gerador de desmontadores e contribuiu com os ajustes necessários nos modelos das CPUs, além de testes no processo de tradução binária. Em função disto, esta seção se limita a mostrar, através de um exemplo, o processo de tradução binária.

A especificação formal do comportamento das instruções da arquitetura está definida na ADL ArchC. Foi criada uma nova propriedade denominada `semantic` para a especificação do comportamento através de operações genéricas básicas (`add`, `sub`, `mult`, `div`, `and`, `or`, `load`, `store`, etc...). A Figura 5.2 exibe a especificação na ADL

ArchC da semântica da instrução `lw` do MIPS-I. O resultado do cálculo do endereço de memória obtido com a operação básica `add` primeiramente é armazenado em uma variável auxiliar `temp`. Na seqüência, é efetuada a leitura do conteúdo referenciado pelo endereço em `temp`, o qual é armazenado em `rt`.

```
1. lw.set_property(semantic, (temp, add, rs, imm));
2. lw.set_property(semantic, (rt, load, temp));
```

Figura 5.2: Especificação formal do comportamento da instrução `lw` do MIPS-I

Para cada instrução descrita no modelo da arquitetura gera-se um grafo contendo apenas operações genéricas. A Figura 5.3 exibe um grafo com as operações genéricas que representam o comportamento da instrução `lw` do MIPS-I.

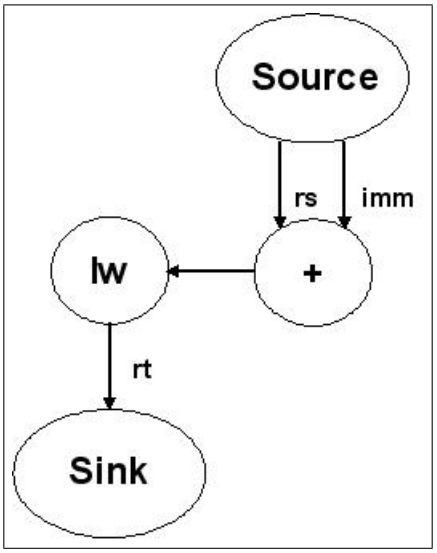

Figura 5.3: Grafo de operações genéricas da instrução `lw` do MIPS-I

O mapeamento de instruções para fins de tradução binária baseia-se em grafos de operações genéricas. Procura-se na arquitetura destino uma instrução ou um conjunto de instruções com comportamento equivalente ao formalizado no grafo. A Figura 5.4 exibe um código MIPS que se deseja traduzir para uma CPU-alvo em que a instrução `lw` correspondente não efetua a soma dos endereços. A Figura 5.5 ilustra os grafos gerados no processo desta tradução e na Figura 5.6 o código final traduzido para a CPU-alvo.

O grafo da Figura 5.5a representa o comportamento de código a ser traduzido (corresponde à Figura 5.4). O grafo da 5.5b representa o comportamento equivalente em termos de operações genéricas (independentes de arquitetura). Por fim, o grafo 5.5c representa o comportamento do códito traduzido (corresponde à Figura 5.6).

```
1. lw $8, 4($5)
2. add $5, $8, $7
```

Figura 5.4: Segmento de código MIPS-I para tradução binária

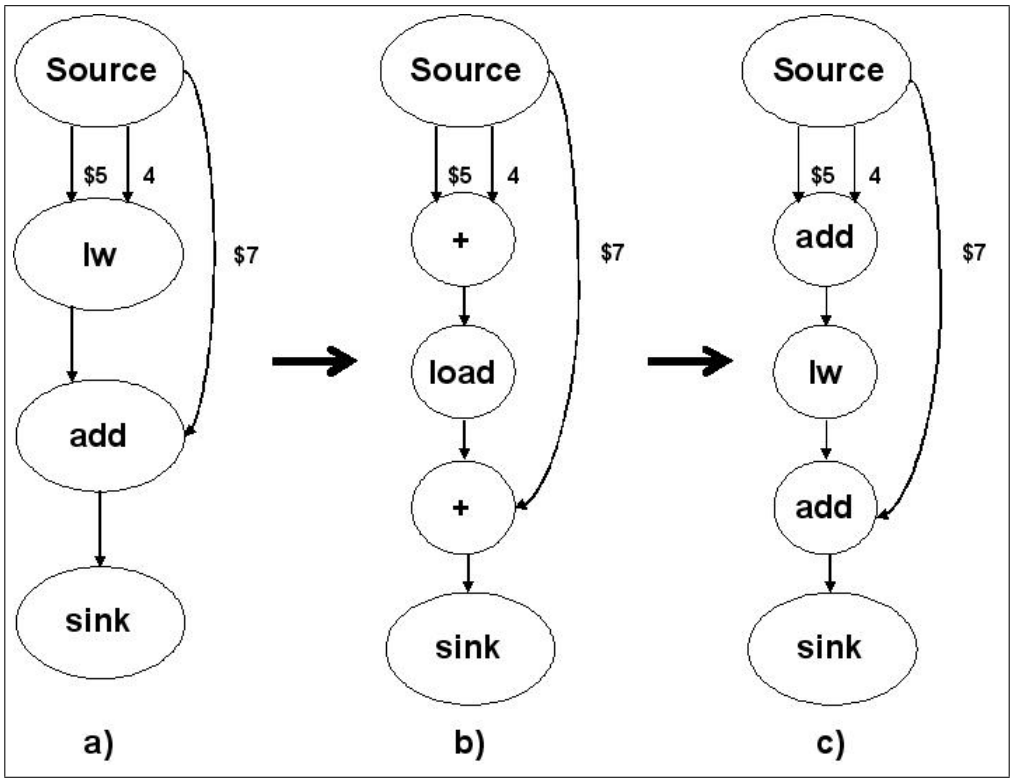

Figura 5.5: Exemplos de grafos gerados no processo de tradução binária

```
1. addi %8, %5, 4
2. lw $9, %8
3. add %5, %9, %7
```

Figura 5.6: Segmento de código obtido da tradução binária

5.3.2 O papel das ferramentas de inspeção de código

No fluxo de tradução binária da Figura 5.1 um gerador de desmontadores é fundamental para viabilizar o processo de tradução binária de forma a se obter o código *assembly* a partir do arquivo-objeto de entrada.

Vale ressaltar que desmontadores convencionais, tais como os gerados com base no pacote *GNU Binutils*, não geram um código *assembly* passível de ser imediatamente submetido a um montador para remontagem. Contudo, isto é resolvido através de arquivos *shell script* do Linux, os quais ajustam o código *assembly* gerado pelo desmontador, harmonizando-o aos requisitos da técnica de tradução binária proposta. Esses ajustes consistem na remoção dos nomes de seção e na conversão dos endereços de memória em *labels* a serem referenciados nas instruções de desvio.

Um gerador de depuradores também é importante no processo de tradução binária visto que o depurador gerado, executado juntamente com um simulador, permite verificar passo-a-passo a completa execução das instruções no arquivo de código objeto disponível para uma CPU 1 como no seu equivalente traduzido para uma CPU 2.

5.4 Resultados experimentais preliminares

Até o momento, o protótipo do tradutor binário proposto restringe-se à tradução binária de blocos básicos e foi validado para as CPUs MIPS e SPARC para o conjunto de *benchmarks* Dalton [DAL 07]. Contudo, o protótipo está sendo estendido para suportar código condicional.

As Tabelas 5.1 e 5.2 mostram os resultados preliminares da tradução do MIPS para o SPARC e do SPARC para o MIPS, respectivamente, listando o número de instruções na arquitetura original e o número de instruções geradas no processo de tradução. Estes números diferem nas duas tabelas em função de algumas instruções da arquitetura de origem não possuírem uma instrução equivalente única na arquitetura destino, sendo então necessária a geração de duas ou mais instruções no processo de tradução.

Note que o número de instruções obtidas via tradução para o MIPS na Tabela 5.2

Tabela 5.1: Resultados da tradução binária MIPS-SPARC

Programa	Número de instruções MIPS	Número de instruções SPARC
cast	21	18
fib	67	85
gcd	31	27
int2bin	16	20
negcnt	12	14
xram	20	29

Tabela 5.2: Resultados da tradução binária SPARC-MIPS

Programa	Número de instruções SPARC	Número de instruções MIPS
cast	25	28
fib	68	78
gcd	29	32
int2bin	15	19
negcnt	08	12
xram	33	40

(terceira coluna) é maior ou igual ao número de instruções geradas pelo compilador para o MIPS na Tabela 5.1 (segunda coluna). Note, entretanto, que o número de instruções obtidas via tradução para o SPARC na Tabela 5.1 (terceira coluna) é menor do que o número de instruções geradas pelo compilador para o SPARC em três casos (`cast`, `gcd` e `xram`), conforme a Tabela 5.2 (segunda coluna). Esta aparente anomalia indica que o compilador MIPS foi mais eficaz na seleção de instruções do que o compilador SPARC. O tradutor binário simplesmente refletiu a melhor qualidade do código de entrada.

A validação mais extensiva da técnica de tradução envolvendo código condicional, outras CPUs e um número maior de programas do *benchmark* será objeto de trabalho futuro (veja Capítulo 6).

Capítulo 6

Conclusão

Nesta seção estão descritas as principais contribuições do trabalho de pesquisa, as ferramentas criadas e as perspectivas de trabalhos futuros.

6.1 Apreciação do trabalho de pesquisa

Os resultados experimentais obtidos com as ferramentas de geração automática de desmontadores e depuradores, baseados na descrição da arquitetura em ADL, evidenciaram funcionalidade apropriada e capacidade de redirecionamento para todos os casos testados, fornecendo assim insumos para o processo de inspeção de código durante a exploração do espaço de projeto para diferentes CPUs alvo.

As ferramentas geradas também contribuem para o fluxo de tradução binária proposto no Capítulo 5, sendo o desmontador o ponto de entrada para o processo de tradução, através da obtenção do código *assembly* do arquivo-objeto original e o depurador podendo ser utilizado no processo de simulação tanto do arquivo-objeto original como do obtido pelo processo de tradução.

6.2 Trabalhos em andamento

- Desenvolvimento de mais experimentos com o fluxo de tradução binária, incorporando mais CPUs e implementando os requisitos que faltam, como suporte a desvios

e instruções que manipulam *flags*.

- Melhorias no código das ferramentas a fim de reduzir os tempos de execução e realização de experimentos com novas CPUs alvo, como Motorola ColdFire e Altera Nios2. Este trabalho ampliará o esquema de validação e permitirá testar ainda mais a capacidade de redirecionamento das ferramentas melhorando o seu desempenho.

6.3 Contribuições técnico-científicas

- A formalização das características inerentes a um desmontador e um depurador fornecendo subsídios importantes para a criação de um gerador para estas ferramentas que funcione com uma ADL arbitrária.

- Uma metodologia de geração automática de ferramentas de inspeção de código no âmbito de desmontagem e depuração de código.

6.4 Produtos de trabalho

A título de resultados das pesquisas relacionadas a esta dissertação foram desenvolvidas e validadas experimentalmente três ferramentas e um modelo, além de artigos submetidos e publicados:

- A modelagem funcional do modo THUMB do processador ARM na ADL ArchC em co-autoria com o mestrando Paulo Kuss.

- Um gerador automático de desmontadores redirecionáveis disponível em domínio público [ARC 07]. Em particular, a técnica utilizada possibilita a geração de um desmontador para um processador que ainda não está portado no GNU, o i8051.

- Um gerador automático de depuradores redirecionáveis cuja previsão para disponibilização em domínio público é entre fevereiro e março de 2007.

- Um protótipo de uma ferramenta de tradução binária para arquivos-objeto entre diferentes arquiteturas, o qual utiliza as ferramentas anteriormente descritas, mais diretamente o gerador de desmontadores. Esse protótipo resultou do trabalho de pesquisa realizado conjuntamente com os mestrandos Daniel Casarotto [CAS 07] e José O. C. Filho [FIL 07].

- Um artigo publicado no XII Workshop IBERCHIP [KUS 06] sobre a modelagem funcional do modo THUMB do processador ARM.

- Um artigo publicado no VI Microeletronics Students Forum [MEN 06b] sobre o gerador de desmontadores.

- Um artigo aceito para publicação no simpósio IEEE ISVLSI 2007 [BAL 07b] e outro submetido ao periódico ACM TODAES [BAL 07a], sobre a geração de ferramentas binárias, como montadores, ligadores, desmontadores e depuradores, baseadas em um modelo abstrato que captura as informações da arquitetura especificadas em ADL, ambos resultantes de pesquisa conjunta UFSC-UNICAMP

- Um artigo aceito para publicação no simpósio SAMOS Workshop 2007 [SCH 07a] e outro submetido ao simpósio IEEE MWSCAS/NEWCAS 2007 [SCH 07b], sobre o gerador de depuradores baseado em um modelo abstrato.

6.5 Tópicos para investigação futura

No desenvolvimento deste trabalho foram identificadas diversas oportunidades de trabalhos futuros co-relacionadas com o trabalho desenvolvido. Dentre tais oportunidades, destacam-se:

- Desenvolvimento de um modelo com precisão de ciclos do modo THUMB do processador ARM, bem como a sua integração com um modelo da própria CPU ARM. Vale ressaltar que um modelo puramente funcional permite apenas avaliar a adequação aos requisitos funcionais e ao tamanho de código, já um modelo com precisão de ciclos viabiliza estimativas de desempenho.

- Desenvolvimento de uma interface gráfica para o depurador a fim de facilitar ao usuário o uso desta ferramenta.

- Avaliação das informações geradas para depuração pelo *cross compiler* do `gcc` através do parâmetro `-g` a fim de adequar o depurador para que consiga inspecionar arquivos-objeto gerados por outros compiladores, pois atualmente somente são simulados arquivos-objeto que tenham sido compilados com este parâmetro.

- Geração das ferramentas binárias de maneira que operem com diferentes formatos de arquivo-objeto, como COFF por exemplo, identificando e gerando os arquivos necessários na biblioteca BFD.

- Elaboração de uma API para o mecanismo de redirecionamento das ferramentas permitindo a geração das ferramentas binárias (montador, ligador, desmontador e depurador) independentemente da ADL adotada, para isso, a formalização definida na Seção 4.5 provê um subsídio importante.

Apêndice A

Utilizando o `acbinutils`

Este anexo apresenta os passos a serem seguidos para a geração dos executáveis do desmontador e do depurador para uma determinada arquitetura descrita em ArchC. São exemplificadas as linhas de comando e descritos os programas utilizados.

Este anexo foi baseado no trabalho de [BAL 05] que desenvolveu a metodologia de redirecionamento do pacote *GNU Binutils* com base na descrição de arquiteturas na ADL ArchC para a geração automática de montadores sendo então agora ajustado o processo de geração para também contemplar a geração automática de desmontadores e depuradores.

Para a geração das ferramentas é necessário que o usuário esteja usando um sistema operacional compatível com o GNU/Linux e com no mínimo as versões abaixo instaladas dos seguintes pacotes (outras versões não foram testadas):

- Bison 2.1 [BIS 07]
- Flex 2.5.4 [FLE 07]
- GCC 3.4.6 [GCC 07]
- GNU Automake 1.9.6 [AUT 07b]
- GNU Autoconf 2.59 [AUT 07a]
- SystemC 2.1 [SYS 07]
- GNU Binutils 2.16.1 [BIN 07]

- GNU Gdb 6.4 [GDB 07]

Como o depurador é executado juntamente com o simulador, os pacotes acima se referem aos requisitos necessários para a geração tanto do simulador como das ferramentas de manipulação de código binário do ArchC, entre elas o desmontador e o depurador. Mais informações sobre a geração de simuladores com suporte a depuração são obtidas no endereço eletrônico do ArchC [ARC 07].

A.1 Gerando as ferramentas binárias

O processo de geração das ferramentas de desmontagem e de depuração de código está embutido no fluxo de geração das demais ferramentas de manipulação binária do ArchC, o `acbinutils`. A geração das ferramentas consiste em três fases:

- Compilar o pacote ArchC;
- Gerar e configurar as funções e arquivos dependentes de arquitetura, e
- Construir o desmontador a partir da estrutura do `Binutils` e o depurador a partir da estrutura do `Gdb`.

A.1.1 Compilando o pacote ArchC

Com o pacote ArchC em mãos [ARC 07], deve-se descompactá-lo em um diretório qualquer. Acessando o diretório raiz da distribuição pela execução do *script* `boot.sh`, serão gerados alguns arquivos necessários à instalação do ArchC, entre eles o `configure`. Executando o `configure`, o usuário tem três opções de configuração, que devem ser passadas como parâmetros: `--with-systemc` = <diretório do pacote SystemC (necessário para geração de simuladores)>, `--with-binutils` = <diretório do pacote *GNU Binutils*>, `--with-gdb` = <diretório do pacote *GNU Debugger*> e de forma opcional o diretório de instalação das ferramentas pelo parâmetro `--prefix`.

Na seqüência, o pacote é compilado através do comando `make` e depois deve-se digitar `make install` para que o mesmo seja instalado no diretório indicado. Após este

```
- archc (raiz)
+ acstone        // exemplos distribuídos junto com o pacote
- autom4te.cache // arquivos gerados pelo autoconf
- bin            // ferramentas executáveis
- config         // arquivos de configuração
- doc            // arquivos de documentação
- etc            // arquivo de configuração
+ include        // arquivos de inclusão utilizados por ferramentas geradas
- lib            // bibliotecas utilizadas pelas ferramentas geradoras e/ou geradas
+ models         // possível diretório de armazenamento dos arquivos de modelos ArchC,
                    o usuário é livre, porém, para usar qualquer outro
- share          // scripts específicos
+ src            // conjunto de arquivos fontes ArchC
```

Figura A.1: Árvore de diretórios do pacote ArchC

processo tanto o gerador de simuladores como o gerador de ferramentas de manipulação binária estarão criados. Segue um exemplo via comandos:

```
./configure --with-systemc = <diretório do systemc> --with-binutils = <diretório do binutils>
  --with-gdb = <diretório do gdb> --prefix = <diretório ArchC>
make
make install
```

Caso o pacote ArchC já esteja instalado, é necessária a reconstrução das ferramentas através do comando `make clean` antes de um novo `configure`.

A Figura A.1 adaptada de [BAL 05] exibe a estrutura de diretórios do ArchC.

A.1.2 Gerando as ferramentas binárias

A ferramenta `acbinutils` é que gera os arquivos fontes para as ferramentas binárias; inicialmente desenvolvida para a geração de montadores e posteriormente adaptada para também gerar os arquivos necessários ao desmontador e ao depurador conforme relatado no Capítulo 4. Contudo são necessários outros processos, como a construção de um diretório local, concatenação de arquivos e substituição de nomes em alguns arquivos fontes. A automação de todo este processo é feita por um *shell script* denominado `acbingen.sh` localizado no diretório `bin`, após a compilação do pacote ArchC.

A função do *script* consiste na criação dos diretórios necessários, execução do

```
Usage: ../../bin/acbingen.sh [options] <model-file>

Create binary utilities source files and optionally build them.

Options:
 -a<name>   sets the architecture name (if ommitted, it defaults to
            <model-file> without the extension
 -i<dir>    build and install the binary utilities in directory <dir>
            NOTE: <dir> -MUST- be an absolute path
 -c         only create the files, do not copy to binutils tree
 -h         print this help
 -v         print version number

Report bugs and patches to ArchC Team.
```

Figura A.2: Sintaxe e opções de linha de comando do `acbingen.sh`

`acbinutils`, cópia dos arquivos fontes gerados na estrutura de diretórios original de distribuição dos pacotes *GNU Binutils* e *GNU Debugger* e inclusive a compilação dos pacotes para criação das ferramentas executáveis.

Conforme a Figura A.2, pela opção `-help` visualiza-se a sintaxe da linha de comando do *shell script* `acbingen.sh`.

`<model-file>` corresponde ao principal arquivo do modelo ArchC usado na geração dos arquivos fontes. Dentre as opções, no parâmetro `-a` não se deve usar um nome que já exista no pacote de distribuição das ferramentas *GNU Binutils* e *GNU Debugger* para evitar erros na geração das ferramentas. Um parâmetro interessante e útil é o `-i`, caso ativado procederá à compilação e instalação das ferramentas no diretório informado.

As tarefas realizadas pelo script são:

1. Criar um diretório local com uma sub-árvore do Binutils e outra do Gdb;

2. Executar o `acbinutils` para a geração dos arquivos nos diretório criados;

3. Criar arquivos específicos para a arquitetura com base em templates;

4. *Patching* de alguns arquivos de configuração na árvore do Binutils e do Gdb;

5. Cópia dos arquivos gerados pela ferramenta `acbinutils` na estrutura de diretórios do Binutils e do Gdb, e

6. Compilação e instalação das ferramentas de manipulação binária, entre elas o desmontador e o depurador, caso passado o parâmetro `-i`.

   Segue um exemplo via comandos:

```
cd <diretório modelo>
<diretório do acbingen.sh>/acbingen.sh -a <nome da arquitetura>
  -i <diretório de instalação das ferramentas> <modelo ArchC>
```

Este processo resulta na geração das ferramentas binárias baseadas na descrição de uma dada arquitetura (entre elas o desmontador e o depurador) as quais residem no diretório de instalação informado. Mais informações quanto ao uso das ferramentas podem ser obtidas pela invocação dos seus respectivos executáveis, através do parâmetro `-help`.

Apêndice B

A modelagem do modo THUMB do processador ARM

Este anexo descreve a arquitetura (Seção B.1) e a modelagem funcional em ArchC (Seção B.2) para o conjunto de instruções compactas do modo THUMB do processador ARM adotado como estudo de caso dos recursos da ADL ArchC. O modelo do THUMB mostrou-se robusto em face da variedade de experimentos a que foi submetido (Seção B.3) com respaldo inclusive por publicação de artigo em congresso científico [KUS 06]. O desenvolvimento do modelo foi feito em conjunto com o colega de mestrado Paulo Fernando Kuss.

B.1 Arquitetura do conjunto de instruções THUMB

O conjunto de instruções THUMB foi criado para atender à demanda de compactação de código em sistemas embarcados visando diminuição do tamanho e custo do sistema de memória [ARM 07b]. Ele é ser visto como uma forma comprimida de um subconjunto de instruções do processador ARM [ARM 96].

O hardware do processador ARM não executa as instruções THUMB diretamente, estas são dinamicamente descomprimidas e mapeadas para instruções ARM, que são as instruções efetivamente executadas, sem perda de desempenho [SLO 04].

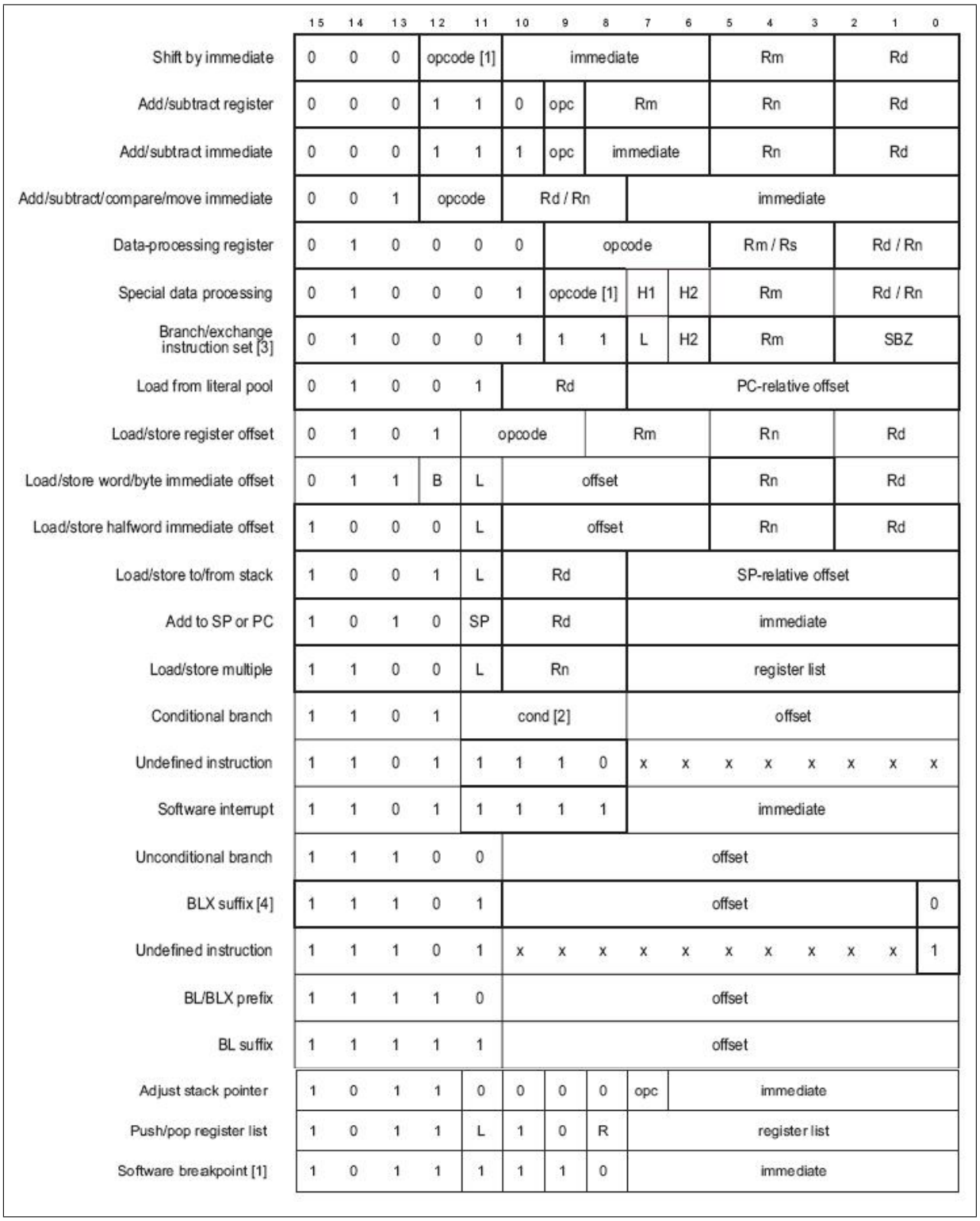

Figura B.1: Conjunto de instruções THUMB

Processadores ARM (com suporte apropriado) interpretam as instruções segundo os formatos de 16 e 32 *bits*, caracterizando os modos de operação THUMB e ARM, respectivamente. A distinção é feita através de um *bit* do registrador de status (o *bit* T do registrador `CPSR`), 1 para modo THUMB e 0 para modo ARM.

Se a CPU estiver operando em modo THUMB, executa-se uma transição explícita para o modo ARM através da instrução *Branch and Exchange* (`BX`). Ademais, uma transição implícita para o modo ARM ocorre sempre que houver uma exceção, pois estas são sempre tratadas no modo ARM.

O programador acessa diretamente oito registradores de uso geral no modo THUMB (R0 a R7), assim como o *program counter* (`PC`), o *stack pointer* (`SP`), o *link register* (`LR`), e o registrador de status (`CPSR`). Os registradores `R8` a `R16` não fazem parte do conjunto de registradores padrão no modo THUMB, mas podem ser usados para armazenamento temporário. O modo THUMB possui 37 instruções codificadas em 25 formatos distintos perfazendo um total de 63 variações possíveis. Os diferentes formatos são exibidos na Figura B.1 que foi adaptada do manual da CPU ARM [ARM 96].

B.2 Descrição funcional em ArchC

A descrição completa do modelo do modo THUMB do processador ARM é extensa, portando as figuras desta seção são versões condensadas da descrição real para fins de visualização de como se organiza seu conteúdo.

Uma descrição de uma arquitetura em ArchC é dividida em duas partes: a descrição do conjunto de instruções da arquitetura (`AC_ISA`) e a descrição dos elementos da arquitetura (`AC_ARCH`).

Na descrição `AC_ARCH` conforme Figura B.2, o projetista descreve os componentes principais da arquitetura, como o tamanho da palavra (Linha 3), os módulos e estruturas de armazenamento (Linhas 5 e 6) e *endian* da arquitetura (Linha 10).

Na descrição `AC_ISA` vizualizada na Figura B.3, o projetista fornece informações sobre o conjunto de instruções (Seção B), tais como formatos e tamanhos (Seção A), mapeamento simbólico ou o respectivo número de cada registrador (Seção C), sintaxe

```
 1. AC_ARCH(thumbv1) {
 2.
 3.   ac_wordsize 32;
 4.
 5.   ac_mem DM:10M;
 6.   ac_regbank RB:18;
 7.
 8.   ARCH_CTOR(thumbv1){
 9.     ac_isa("thumbv1_isa.ac");
10.     set_endian("little");
11.   };
12. };
```

Figura B.2: Modelo do THUMB: elementos da arquitetura

assembly e códigos operacionais (Seção D).

A partir destas duas descrições, o pré-processador ArchC (`acpp`), que é composto por analisadores léxico, sintático e semântico gerados pelas ferramentas GNU Flex [FLE 07] e GNU Bison [BIS 07], gera o esqueleto do simulador da arquitetura, que contém as assinaturas dos métodos que especificam o comportamento de cada instrução em linguagem C/C++ ilustrado na Figura B.4. Em seguida, define-se o comportamento comum a todas as instruções (Linhas 2 a 5), um comportamento específico para cada formato de instrução (Linhas 8 a 9) e o comportamento específico de cada instrução (Linhas 12 a 32) [ARC 07].

Exemplificando o formato `ASR` da Figura B.3 (Linha 3), tem-se 3 instruções associadas ao mesmo (Linha 9), `add3`, `mov2` e `sub3`. A sintaxe *assembly* da instrução `add3` por exemplo está definida na linha 27 e o seu código operacional na linha 28. Na Figura B.4 (Linhas 12 a 32) está implementado o seu comportamento, ou seja, uma soma envolvendo o conteúdo de dois registradores (`rnASR` e `rmASR`) e gravando o resultado no registrador `rdADR` (Linhas 17, 18 e 28) além de setar flags de controle (Linhas 20 a 26).

A descrição desenvolvida e o respectivo modelo executável permitem a simulação de qualquer programa que contenha apenas instruções THUMB. Assim, o modelo não suporta programas que façam transições do modo THUMB para o modo ARM. Esta limi-

```
1. AC_ISA(thumbv1){
2.   //Seção A - Declaração dos formatos de instrução
3.   ac_format Type_ASR = "%rdASR:3 %rnASR:3 %rmASR:3 %opcASR:1 %funcASR:6";
4.   ac_format Type_CB = "%immCB:8 %condCB:4 %funcCB:4";
5.   ac_format Type_ASPPC = "%immASPPC:8 %rdASPPC:3 %spASPPC:1 %funcASPPC:4";
6.   ac_format Type_MISC3 = "%immMISC3:8 %funcMISC3:8";
7.
8.   //Seção B - Mapeamento de instruções para formatos
9.   ac_instr<Type_ASR> add3, mov2, sub3;
10.  ac_instr<Type_CB> b1;
11.  ac_instr<Type_ASPPC> ldr4, str3;
12.  ac_instr<Type_MISC3> bkpt;
13.
14.  //Seção C - Mapeamento dos registradores
15.  ac_asm_map reg {
16.    "$"[0..17] = [0..17];
17.    "r"[0..12] = [0..12];
18.    "$sp" = 13;
19.    "$lr" = 14;
20.    "$pc" = 15;
21.    "$cpsr" = 16;
22.    "$spsr" = 17;
23.  }
24.
25.  //Seção D - Mapeamento de mnemônicos e códigos operacionais
26.  ISA_CTOR(thumbv1){
27.    add3.set_asm("add %reg, %reg, %reg", rdASR, rnASR, rmASR);
28.    add3.set_decoder(opcASR=0x00, funcASR=0x06);
29.
30.    mov2.set_asm("mov %reg, %reg", rdASR, rnASR);
31.    mov2.set_decoder(opcASR=0x00, funcASR=0x07, rmASR=0x00);
32.
33.    sub3.set_asm("sub %reg, %reg, %reg", rdASR, rnASR, rmASR);
34.    sub3.set_decoder(opcASR=0x01, funcASR=0x06);
35.
36.    b1.set_asm("b %exp %exp", condCB, immCB);
37.    b1.set_decoder(funcCB=0x0D);
38.
39.    ldr4.set_asm("ldr %reg, [ $sp, %exp * 4 ]", rdASPPC, immASPPC);
40.    ldr4.set_decoder(spASPPC=0x01, funcASPPC=0x09);
41.
42.    str3.set_asm("str %reg, [ $sp, %exp * 4 ]", rdASPPC, immASPPC);
43.    str3.set_decoder(spASPPC=0x00, funcASPPC=0x09);
44.
45.    bkpt.set_asm("bkpt %exp", immMISC3);
46.    bkpt.set_decoder(funcMISC3=0xBE);
47.  };
48. };
```

Figura B.3: Modelo do THUMB: conjunto de instruções

```
1. //Seção A - Comportamento genérico para todas as instruções
2. void ac_behavior( instruction ) {
3.   dbg_printf("----- PC=%#x ----- %lld\n", (int) ac_pc, ac_instr_counter);
4.   ac_pc += 2;
5. };
6.
7. //Seção B - Comportamento específico para cada formato de instrução
8. void ac_behavior( Type_ASR ) {
9. }
10.
11. //Seção C - Comportamento específico para cada instrução
12. void ac_behavior( add3 ) {
13.   long long alu_out;
14.   dbg_printf("add r%d, r%d, r%d\n", rdASR, rnASR, rmASR);
15.   dbg_printf("add3 %#x, %#x, %#x\n", RB[rdASR], RB[rnASR], RB[rmASR]);
16.
17.   alu_out = (unsigned long long)(unsigned long)RB[rnASR] +
18.             (unsigned long long)(unsigned long)RB[rmASR];
19.
20.   flags.N = getBit(alu_out,31);
21.   flags.Z = ((alu_out == 0) ? true : false);
22.   flags.C = (alu_out > 0xFFFFFFFF ? true : false);
23.   flags.V = (((getBit(RB[rnASR],31) && getBit(RB[rmASR],31) &&
24.               (!getBit(alu_out,31))) ||
25.              ((!getBit(RB[rnASR],31)) && (!getBit(RB[rmASR],31)) &&
26.                 getBit(alu_out,31))) ? true : false);
27.
28.   RB[rdASR] = alu_out;
29.
30.   dbg_printf("Result: %#x\nFlags: N=%d, Z=%d, C=%d, V=%d\n",
31                RB[rdASR],flags.N,flags.Z,flags.C,flags.V);
32. };
```

Figura B.4: Modelo do THUMB: comportamento das instruções

tação será eliminada quando este modelo for integrado ao modelo do ARM5, em fase de desenvolvimento.

B.3 Validação experimental do modelo

Os experimentos foram executados em um computador com CPU Intel® Pentium 4 (1.8 GHz), com 256 MB de memória principal.

O sistema operacional utilizado foi o Mandrake GNU/Linux. Para a geração do simulador, utilizou-se a ADL ArchC, versão 1.5.1. Utilizou-se também o *cross-compiler* ARM GCC versão 3.3.1 [ARM 07a] que, através dos parâmetros `-EL -mthumb`, determina a configuração *little endian* e força a geração de código somente com instruções THUMB, tanto no formato *assembly* (para inspeção da sintaxe das instruções geradas) quanto no formato binário (para interpretação pelo simulador gerado pela ferramenta `acsim` do pacote ArchC [ARC 07]).

Há que se fazer uma ressalva. Como o modelo proposto representa somente o modo THUMB (a ser oportunamente integrado com trabalho correlato focado no modo ARM), há uma dificuldade operacional em se modelar o término de um programa. Na prática, a CPU é transicionada do modo THUMB para o modo ARM, através da instrução `BX` (*Branch and Exchange*). Assim, foi utilizada uma solução provisória: a instrução `BX` é codificada para forçar o término da simulação, sem perda de generalidade para a modelagem do modo THUMB. Obviamente, esta limitação será removida quando os modelos de ambos os modos forem integrados.

Os tempos de simulação associados aos *benchmarks* foram obtidos através de estatísticas geradas a partir do conjunto de ferramentas de ArchC, que são expressos em unidades de tempo de SystemC (*default time units*) [SYS 07].

Inicialmente, cada instrução foi testada e depurada isoladamente. Um programa *assembly* contendo todas as instruções do THUMB foi manualmente desenvolvido, montado e simulado para fins de depuração preliminar do modelo. Em seguida, uma validação mais ampla foi realizada através de experimentos baseados em programas extraídos do *benchmark* do Projeto Dalton [DAL 07]. Cada programa do *benchmark* foi compilado conforme descrito anteriormente. O código objeto resultante foi alimentado no simulador gerado pela ferramenta `acsim` [ARC 07], a partir da descrição funcional do modo THUMB.

Tabela B.1: Resultados para os programas do *benchmark*

Programa	**Instruções Executadas**	**Tempo de Simulação**	**Tamanho de Código (*bytes*)**
negcnt	168	0.12	34489
gcd	242	0.21	34515
int2bin	189	0.14	34483
cast	96	0.06	34568
divmul	304	0.23	34562
fib	602	0.57	34681
sort	2388	2.16	36558
xram	4244	4.13	34586
sqroot	1379	1.23	133435

A Tabela B.1 apresenta os resultados obtidos na simulação correta de cada um dos programas do *benchmark*.

Referências bibliográficas

[ABB 02] ABBASPOUR, M.; ZHU, J. Retargetable binary utilities. In: PROCEEDINGS OF THE 39TH CONFERENCE ON DESIGN AUTOMATION, 2002. ACM Press, 2002. p.331–336.

[ALT 00] ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the opportunities of binary translation. **Computer**, [S.l.], v.33, n.3, p.40–45, March, 2000.

[ARC 07] ARCHC. **The ArchC Website**. Disponível em <http://www.archc.org>. Acesso em: 22 jan. 2007.

[ARM 96] ARM. **ARM architecture reference manual**. ARM Limited, 1996. ARM DDI0100E.

[ARM 07a] ARM. **ARM Linux Toolchains**. Disponível em <http://ftp.arm.linux.org.uk/pub/armlinux/toolchain/>. Acesso em: 22 jan. 2007.

[ARM 07b] ARM. **ARM, The Architecture for the Digital World**. Disponível em <http://www.arm.com>. Acesso em: 22 jan. 2007.

[AUT 07a] AUTOCONF. **The GNU Autoconf Website**. Disponível em <http://www.gnu.org/software/autoconf>. Acesso em: 22 jan. 2007.

[AUT 07b] AUTOMAKE. **The GNU Automake Website**. Disponível em <http://www.gnu.org/software/automake>. Acesso em: 22 jan. 2007.

[BAC 79] BACKUS, J.; NAUR, P. Appendix D of the CDC Algol-60 reference manual. Control Data Corporation, 1979. Relatório Técnico Versão 5.

[BAL 05] BALDASSIN, A. **Geração automática de montadores em ArchC**. 2005. 95 p. Dissertação (Mestrado em Ciência da Computação): Instituto de Computação, UNICAMP, Campinas, 2005.

[BAL 07a] BALDASSIN, A. et al. An ESL-Compliant Binary Utility Generator. **Submetido ao ACM Transactions on Design Automation of Electronic Systems**, [S.l.], v., 2007.

[BAL 07b] BALDASSIN, A. et al. Automatic Retargeting of Binary Utilities for Embedded Code Generation. 2007. Aceito no IEEE Computer Society Annual Symposium on VLSI, 2007.

[BIN 07] BINUTILS, G. **The GNU Binutils Website**. Disponível em <http://www.gnu.org/software/binutils>. Acesso em: 22 jan. 2007.

[BIS 07] BISON. **The Bison Parser Generator Website**. Disponível em <http://www.gnu.org/software/bison>. Acesso em: 22 jan. 2007.

[CAS 06] CASAROTTO, D.; DOS SANTOS, L. C. V. Automatic link editor generation for embedded CPU cores. In: PROCEEDINGS OF THE 4TH INTERNATIONAL IEEE NORTHEAST WORKSHOP ON CIRCUITS AND SYSTEMS, 2006. IEEE Computer Society, 2006. p.121–124.

[CAS 07] CASAROTTO, D. C. **Utilitários Binários Redirecionáveis: da Linkedição rumo à Tradução Binária**. 2007. 83 p. Dissertação (Mestrado em Ciência da Computação): Departamento de Informática e Estatística, UFSC, Florianópolis, 2007.

[CHA 91] CHAMBERLAIN, S. **Libbfd, the binary file descriptor library**. Free Software Foundation, Inc., April, 1991. version 3.0.

[CHE 97] CHERNOFF, A.; HOOKWAY, R. DIGITAL FX!32 — running 32-Bit x86 applications on alpha NT. In: Proceedings of the USENIX Windows NT Workshop, 1997. [s.n.], 1997. p.9–13.

[CIF 02] CIFUENTES, C. et al. Experience in the design, implementation and use of a retargetable static binary translation framework. Sun Labs Tech Report, January, 2002. Relatório Técnico TR-2002-105.

[COW 07] COWARE. **LisaTek**. Disponível em <http://www.coware.com>. Acesso em: 22 jan. 2007.

[DAL 07] DALTON, P. **Synthesizable VHDL Model of 8051**. Disponível em <http://www.cs.ucr.edu/ dalton/i8051/i8051syn>. Acesso em: 22 jan. 2007.

[FAU 93] FAUTH, A.; KNOLL, A. Automated generation of DSP program development tools using a machine description formalism. In: PROCEEDINGS OF THE IEEE ICASSP-93, 1993. [s.n.], 1993. p.457–460.

[FAU 95] FAUTH, A.; Van Praet, J.; FREERICKS, M. Describing instruction set processors using nML. In: PROCEEDINGS OF THE EDTC: THE EUROPEAN DESIGN AND TEST CONFERENCE, 1995. IEEE Computer Society, 1995. p.503–507.

[FIL 07] FILHO, J. O. C. **Escalonamento Redirecionável de Código sob Restrições de Tempo Real**. 2007. 93 p. Dissertação (Mestrado em Ciência da Computação): Departamento de Informática e Estatística, UFSC, Florianópolis, 2007.

[FLE 07] FLEX. **The Flex Lexical Analyzer Generator Website**. Disponível em <http://www.gnu.org/software/flex>. Acesso em: 22 jan. 2007.

[FRE 93] FREERICKS, M. The nML machine description formalism. Technical University of Berlin, July, 1993. Relatório Técnico Draft, version 1.5.

[GCC 07] GCC. **The GNU Compiler Collection Website**. Disponível em <http://gcc.gnu.org>. Acesso em: 22 jan. 2007.

[GDB 07] GDB, G. **The GNU Debugger Website**. Disponível em <http://www.gnu.org/software/gdb>. Acesso em: 22 jan. 2007.

[GHE 05] GHENASSIA, F. **Transaction-Level Modeling with SystemC - TLM Concepts and Applications for Embedded Systems**. Dordrecht: Springer, 2005. 271 p.

[GUT 01] GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: PROCEEDINGS OF THE 4TH ANNUAL IEEE WORKSHOP ON WORKLOAD CHARACTERIZATION, 2001. [s.n.], 2001. p.3–14.

[HAD 97] HADJIYIANNIS, G.; HANONO, S.; DEVADAS, S. ISDL: an instruction set description language for retargetability. In: PROCEEDINGS OF THE 34TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 1997. ACM Press, 1997. p.299–302.

[HAD 98] HADJIYIANNIS, G. **ISDL: instruction set description language - version 1.0**. MIT Laboratory for Computer Science, November, 1998.

[HAR 97] HARTOOG, M. R. et al. Generation of software tools from processor descriptions for hardware/software codesign. In: PROCEEDINGS OF THE 34TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 1997. ACM Press, 1997. p.303–306.

[HOF 01] HOFFMANN, A. et al. Generating production quality software development tools using a machine description language. In: PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2001. IEEE Press, 2001. p.674–678.

[HOF 02] HOFFMANN, A.; MEYR, H.; LEUPERS, R. **Architecture exploration for embedded processors with LISA**. Kluwer Academic Publishers, 2002.

[JAI 99] JAIN, N. C. **Disassembler using high level processor models**. Dissertação de Mestrado: Department of CSE, IIT, Kanpur, 1999.

[KäS 00] KäSTNER, D. Propan: A retargetable system for postpass optimizations and analyses. In: PROCEEDINGS OF THE ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILERS, AND TOOLS FOR EMBEDDED SYSTEMS, 2000. ACM Press, 2000. p.63–80.

[KUS 06] KUSS, P. F. et al. Modelagem funcional do modo THUMB do processador ARM. In: PROCEEDINGS OF THE XII WORKSHOP IBERCHIP, 2006. IWS, 2006. p.320–321.

[LEU 01] LEUPERS, R.; MARWEDEL, P. **Retargetable Compiler Technology for Embedded Systems - Tools and Applications**. Dordrecht: Kluwer Academic Publishers, 2001. 175 p.

[MAR 03] MARWEDEL, P. **Embedded System Design**. Dordrecht: Kluwer Academic Publishers, November, 2003. 258 p.

[MEN 06a] MENDONÇA, A. K. I.; CARVALHO, F. G. **Desmontador e Depurador Redirecionáveis para ASIPs**. 2006. 215 p. Monografia (Graduação em Ciência da Computação): Departamento de Informática e Estatística, UFSC, Florianópolis, Agosto, 2006.

[MEN 06b] MENDONÇA, A. K. I. et al. Automatic ADL-based Generation of Disassembling Tools. 2006. VI SBC/SBmicro Microelectronics Students Forum, 2006.

[MOO 00] MOONA, R. Processor models for retargetable tools. In: PROCEEDINGS OF THE 11TH IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 2000. [s.n.], 2000. p.34–39.

[PAT 04] PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/Software Interface**. third. ed. Morgan Kaufmann Publishers, 2004. 656 p.

[PDE 07] PDESIGNER. **PDesigner Framework**. Disponível em <http://www.pdesigner.org>. Acesso em: 22 jan. 2007.

[PEE 99] PEES, S. et al. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In: PROCEEDINGS OF THE 36TH ACM/IEEE CONFERENCE ON DESIGN AUTOMATION, 1999. ACM Press, 1999. p.933–938.

[PES 93] PESCH, R. H.; OSIER, J. M. **The GNU binary utilities**. Free Software Foundation, Inc., May, 1993. version 2.16.1.

[RAJ 98] RAJESH, V. **A generic approach to performance modeling and its application to simulator generator**. Dissertação de Mestrado: Department of CSE, IIT, Kanpur, August, 1998.

[RAJ 99] RAJIV, A. **Retargetable profiling tools and their application in cache simulation and code instrumentation**. Dissertação de Mestrado: Department of CSE, IIT, Kanpur, December, 1999.

[RAM 92] RAMSEY, N.; HANSON, D. R. A retargetable debugger. In: PROCEEDINGS OF THE ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1992. ACM Press, 1992. p.22–31.

[RAM 94] RAMSEY, N.; FERNÁNDEZ, M. New Jersey machine-code toolkit reference manual. Department of Computer Science, Princeton University, October, 1994. Relatório Técnico TR-471-94.

[RAM 97] RAMSEY, N.; FERNÁNDEZ, M. Specifying representations of machine instructions. **ACM Transactions on Programming Language and Systems**, [S.l.], v.19, n.3, p.492–524, May, 1997.

[RIG 04] RIGO, S. **ArchC: Uma linguagem de descrição de arquiteturas**. 2004. 107 p. Tese (Doutorado em Ciência da Computação): Instituto de Computação, UNICAMP, Campinas, 2004.

[SAL 07] SALTO. **SALTO Project**. Disponível em <http://www.irisa.fr/caps/projects/Salto>. Acesso em: 22 jan. 2007.

[SCH 07a] SCHULTZ, M. R. O. et al. A model-driven automatically-retargetable debug tool for embedded systems. 2007. Aceito no SAMOS VII Workshop - International Workshop on Systems, Architectures, Modeling and Simulation, 2007.

[SCH 07b] SCHULTZ, M. R. O. et al. A model-driven automatically-retargetable debug tool for embedded systems. 2007. Submetido ao 50th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS) and the 5th of the IEEE International NEWCAS conference, 2007.

[SEE 07] SEEP. **Platform-based Electronic Embedded Systems**. Disponível em <http://www.inf.ufrgs.br/ lse/fluxo.php>. Acesso em: 22 jan. 2007.

[SLO 04] SLOSS, A. N.; SYMES, D.; WRIGHT, C. **ARM System Developer's Guide - Designing and Optimizing System Software**. Morgan Kaufmann Publishers, 2004. 689 p.

[STA 04a] STALLMAN, R. **Using the GNU compiler collection**. Free Software Foundation, Inc., May, 2004. For GCC version 3.4.3.

[STA 04b] STALLMAN, R. M. et al. **Debugging with GDB, the GNU source-level debugger**. Free Software Foundation, Inc., Ninth. ed., February, 2004.

[STA 07] STANDARDS, T. I. **Executable and Linking Format (ELF) Specification**. Disponível em <http://x86.ddj.com/ftp/manuals/tools/elf.pdf>. Acesso em: 22 jan. 2007.

[SV 01] SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design and Test of Computers**, [S.l.], v.18, n.6, p.23–33, November/December, 2001.

[SYS 07] SYSTEMC. **The Open SystemC Initiative**. Disponível em <http://www.systemc.org>. Acesso em: 22 jan. 2007.

[TAG 05] TAGLIETTI, L. **Geração automática de ferramentas de suporte ao desenvolvimento de software embarcado para ASIPs**. 2005. 76 p. Dissertação (Mestrado em Ciência da Computação): Departamento de Informática e Estatística, UFSC, Florianópolis, 2005.

[TEC 07] TECHNOLOGIES, T. C. **Accelerating the design of flexible and royalty-free IP cores**. Disponível em <http://www.retarget.com>. Acesso em: 22 jan. 2007.

[ZIV 96] ZIVOJNOVIC, V.; PEES, S.; MEYR, H. LISA – machine description language and generic machine model for HW/SW co-design. In: PROCEEDINGS OF THE IEEE WORKSHOP ON VLSI SIGNAL PROCESSING, 1996. [s.n.], 1996. p.127–136.